



TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

LAURI VÄLIMAA
MQTT CLIENT IMPLEMENTATION IN IEC 61131-3 COMPATIBLE
PROGRAMMING ENVIRONMENT

Master of Science Thesis

Examiners: Professor José L. Martínez Lastra, Senior Research Fellow
Jani Jokinen
Examiners and topic approved by the
Faculty Council of the Engineering
Sciences on 4 January 2017

ABSTRACT

LAURI VÄLIMAA: MQTT client implementation in IEC 61131-3 compatible programming environment

Tampere University of Technology

Master of Science Thesis, 60 pages, 0 Appendix pages

January 2017

Master's Degree Programme in Automation Technology

Major: Factory Automation and Industrial Informatics

Examiners: Professor José L. Martínez Lastra, Senior Research Fellow Jani Jokinen

Keywords: MQTT, CODESYS, IoT

Collecting data from various sources is essential for systems which leverage the ideology and advantages behind Internet of Things (IoT). Expanding device support in such systems is crucial since it allows collecting data on wider scale but also from more diverse sources. Expanding device support is especially important in terms of novel IoT applications that would require data which just is not yet available.

Major goal in this work is to expand device support of existing IoT system. System in question is IoT-Ticket which is a commercial product developed by Wapice Ltd. It enables remote monitoring and control of connected devices in addition to analyzing of collected data and automatic reporting via web-based user interface.

In this work IoT-Ticket's device support is to be expanded so that CODESYS (COntroller DEvelopment SYstem) compatible devices can be used as valid data sources for this system in the future. CODESYS enabled devices are primarily programmable logic controllers (PLC) that are used to control various industrial processes in different environments. CODESYS itself is a manufacturer independent programming environment which is compatible with IEC 61131-3 standard.

Support for CODESYS enabled devices was required to be developed using MQTT (Message Queuing Telemetry Transport) protocol. MQTT is open application level protocol based on publish-subscribe messaging pattern. In addition to this, it was also required that the data which is sent to IoT-Ticket must be encrypted when it is transferred over unsecure communication channel. Both of these requirements were achieved with MQTT client which was developed for CODESYS in this work. Data encryption in developed client was implemented using symmetric encryption.

TIIVISTELMÄ

LAURI VÄLIMAA: MQTT-asiakasohjelmatoteutus IEC 61131-3 -standardin mukaisessa ohjelmointiympäristössä

Tampereen teknillinen yliopisto

Diplomityö, 60 sivua, 0 liitesivua

Tammikuu 2017

Automaatiotekniikan diplomi-insinöörin tutkinto-ohjelma

Pääaine: Factory Automation and Industrial Informatics

Tarkastajat: professori José L. Martínez Lastra, yliopistotutkija Jani Jokinen

Avainsanat: MQTT, CODESYS, IoT

Datan kerääminen erilaisista lähteistä on keskeistä järjestelmissä, jotka perustuvat asioiden Internetin (engl. Internet of Things, IoT) taustalla olevaan ideologiaan ja sen tarjoamiin hyötyihin. Tällaisissa järjestelmissä laitetuen kasvattaminen on keskeisessä roolissa, sillä se mahdollistaa datan keräämisen laajemmassa mittakaavassa, mutta myös monipuolisemmin erilaisista lähteistä. Laitetuen kasvattaminen on tärkeää erityisesti uudenlaisten IoT-sovellusten kannalta, joiden tarvitsemaa dataa ei yksinkertaisesti ole vielä saatavilla.

Tämän työn keskeinen tavoite on laajentaa olemassa olevan IoT-järjestelmän laitetukea. Kyseinen järjestelmä on Wapice Oy:n kehittämä kaupallinen IoT-Ticket-niminen tuote. Se mahdollistaa järjestelmään liitettyjen laitteiden etäseurannan ja -ohjauksen sekä kerätyn datan analysoinnin ja automaattisen raportoinnin web-pohjaisen käyttöliittymän kautta.

IoT-Ticketin laitetukea tullaan laajentamaan tämän työn myötä siten, että CODESYS (CONtroller DEvelopment SYStem) -yhteensopivia laitteita voidaan käyttää datalähteinä järjestelmälle tulevaisuudessa. CODESYS-yhteensopivat laitteet ovat pääasiassa ohjelmoitavia logiikoita (engl. Programmable Logic Controller, PLC), joita käytetään teollisten prosessien ohjauksessa erilaisissa ympäristöissä. CODESYS itsessään on valmistajariippumaton IEC 61131-3 -standardin mukainen ohjelmointiympäristö.

Vaatimuksena oli toteuttaa CODESYS-laitetuki käyttäen MQTT (Message Queuing Telemetry Transport) -protokollaa. MQTT on avoin sovellustason protokolla, joka perustuu julkaise-tilaa-kommunikointimalliin (engl. publish-subscribe pattern). Vaatimuksena oli lisäksi salata IoT-Tickettiin lähetettävä data, kun sitä siirretään suojaamattoman kommunikointikanavan yli. Molemmat näistä vaatimuksista täyttyivät MQTT-asiakasohjelmalla, joka kehitettiin CODESYS:lle tässä työssä. Datan salaaminen kehitetyssä asiakasohjelmassa toteutettiin symmetrisellä salausmenetelmällä.

PREFACE

This Master of Science Thesis has been made for Wapice Ltd. during 2016. I would like to express my thanks for my supervisor Jani Jokinen from TUT (Tampere University of Technology) for providing guidance for the writing process. I am also thankful for Samuli Reponen and Risto Pajula from Wapice for making this thesis possible.

Special thanks belong to my family for supporting and encouraging me through my studies in TUT. Last but not least, I would also express sincere thanks for all of my friends.

Tampere, 14.1.2017

Lauri Välimaa

TABLE OF CONTENTS

1.	INTRODUCTION	1
1.1	Motivation, objective and scope of the research	1
1.2	Research approach and thesis outline.....	2
2.	THEORETICAL BACKGROUND.....	3
2.1	Internet of Things	3
2.1.1	General architecture and common elements	4
2.1.2	Wapice IoT-Ticket	6
2.2	Security fundamentals	8
2.2.1	Symmetric and asymmetric cryptography	8
2.2.2	RC4 stream cipher.....	10
2.2.3	Message authentication	12
2.2.4	SHA-256 algorithm.....	14
2.2.5	Overview of TLS and X.509 certificates	17
3.	MQTT PROTOCOL	21
3.1	Overview	21
3.2	Publish-subscribe pattern	21
3.3	Main features and functionality.....	22
3.3.1	Connection establishment	22
3.3.2	Publish and subscribe messages, topics	23
3.3.3	Quality of service	24
3.3.4	Security	25
3.4	Control packets.....	26
3.4.1	CONNECT	27
3.4.2	CONNACK.....	29
3.4.3	PUBLISH	30
3.4.4	DISCONNECT	30
4.	CODESYS	32
4.1	Overview	32
4.2	Components.....	32
4.3	Programming.....	33
4.3.1	IEC 61131-3 standard and programming languages.....	34
4.3.2	Socket programming API.....	36
5.	IMPLEMENTATION	39
5.1	Overview of the system.....	39
5.2	CODESYS MQTT client	40
5.2.1	Features	40
5.2.2	Usage.....	41
5.2.3	Architecture.....	44
5.2.4	Improvements.....	45
5.3	MQTT broker	47

5.4	MQTT-REST adapter.....	48
5.5	System configurations	50
6.	CONCLUSIONS.....	53
7.	REFERENCES.....	55

ABBREVIATIONS AND SYMBOLS

3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
API	Application Programming Interface
ARM	Advanced Reduced Instruction Set Computer Machines
CA	Certificate Authority
CAN	Controller Area Network
CMAC	Cipher-based Message Authentication Code
CODESYS	COntroller DEvelopment SYStem
CSV	Comma-Separated Values
DES	Data Encryption Standard
DHE	Ephemeral Diffie-Hellman
E2E	End-to-End
ECDHE	Ephemeral Elliptic curve Diffie-Hellman
FB	Function Block
FBD	Function Block Diagram
FIPS	Federal Information Processing Standards
GCM	Galois/Counter Mode
HMI	Human Machine Interface
HMAC	Hash-based Message Authentication Code
HTTP	Hypertext Transport Protocol
HTTPS	Hypertext Transfer Protocol Secure
I/O	Input/Output
IDE	Integrated Development Environment
IEC 61131-3	International Electrotechnical Commission 61131-3
IL	Instruction List
IoT	Internet of Things
IPv4	Internet Protocol version 4
JSON	JavaScript Object Notation
LD	Ladder Diagram
M2M	Machine to Machine
MAC	Message Authentication Code
MQTT	Message Queuing Telemetry Transport
NIST	National Institute of Standards and Technology
OASIS	Organization for the Advancement of Structured Information Standards
OPC	Object Linking and Embedding for Process Control
OPC UA	Object Linking and Embedding for Process Control Unified Architecture
PLC	Programmable Logic Controller
POU	Program Organization Unit
QoS	Quality of Service
RC4	Rivest Cipher 4
REST	Representational State Transfer
RFID	Radio Frequency Identification
RS-232	Recommended Standard 232
RS-485	Recommended Standard 485
RSA	Rivest-Shamir-Adleman
SFC	Sequential Function Chart

SHA	Secure Hash Algorithm
SSL	Secure Sockets Layer
ST	Structured Text
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TLS	Transport Layer Security
URL	Uniform Resource Locator
UTF	Universal Character Set Transformation Format
XML	Extensible Markup Language
XOR	Exclusive OR
\oplus	Bitwise exclusive OR operation
\parallel	Concatenation operation
\wedge	Bitwise AND operation
\vee	Bitwise OR operation
\neg	Bitwise complement operation
\gg	Right shift operation
\ll	Left shift operation

1. INTRODUCTION

During the past decade, the number of Internet enabled devices has continuously increased. In addition to traditional computers and mobile devices, everyday objects such as domestic appliances are also getting started to be connected to the Internet. The technological development has made it feasible to equip them with hardware which transform them from traditional offline objects to Internet capable devices that are able to communicate and interact with other connected devices. Such development is also referred as next evolution of the Internet which is better known as Internet of Things (IoT).

The principal idea behind IoT is about of connecting various objects to the Internet and making their data available for other networked systems to be utilized. Collecting and refining data from various sources offers new possibilities to develop novel applications which were not previously possible when such data was simply not available. Therefore, having access to the data from various sources is crucial in terms of IoT applications. That requires suitable communication interfaces and interoperability between different components which take part in data delivery process from source to end system. In the process, security is also major concern since transferred data is usually sensitive and thus is not allowed to be sent in plaintext over unsecure communication channel. As such, various solutions exist and are being continuously developed for making data sources securely available for networked IoT applications.

1.1 Motivation, objective and scope of the research

Collecting data from various sources is essential for systems which leverage the ideology and advantages behind Internet of Things. Supporting different types of devices is important since it opens new possibilities to collect, analyze and utilize more but also more diverse data from various sources. This to be possible, the data sources (devices) must be equipped with suitable communication interfaces so their data can be accessed over the network.

The objective of this work is essentially to improve data source support of particular IoT system by implementing support for devices that are not currently supported. The IoT system in question is a commercial product known as IoT-Ticket. It is an IoT platform developed by Wapice Ltd.

IoT-Ticket's device support is expanded in this work so that CODESYS (COntroller DEvelopment SYStem) compatible devices could be used as valid data sources for the system in the future. The principal goal in this work is to make possible to transfer data from

CODESYS compatible device to IoT-Ticket. CODESYS enabled devices are primarily PLCs (Programmable Logic Controllers) that are used to control diverse industrial processes in different environments.

The support for CODESYS enabled devices was required to be implemented using MQTT (Message Queuing Telemetry Transport) protocol. It was also required that the data sent to IoT-Ticket must be encrypted when it is sent over unsecure communication channel.

1.2 Research approach and thesis outline

In this work research was largely carried out in form of practical work by developing MQTT client library for CODESYS. Significant part of the development composed of programing (writing, testing and debugging of source code) in CODESYS. The development work required also studying relevant literature in great extent. This was especially the case with MQTT protocol and CODESYS network programming. Studying cryptography and information security had also major role in this work. They were studied in necessary extent so that required security features for the MQTT client could be implemented.

This work is organized as follows. Chapter 2 covers theoretical background needed for the work. It provides introduction to IoT, Wapice IoT-Ticket, symmetric and asymmetric cryptography, message authentication and TLS (Transport Layer Security). Chapter 3 presents the MQTT protocol itself and structure of relevant MQTT control packets. Chapter 4 briefly presents CODESYS in necessary extend focusing especially its programming aspects. Chapter 5 covers details regarding the actual CODESYS MQTT client implementation done in this work. Chapter 6 contains conclusions.

2. THEORETICAL BACKGROUND

This chapter acts as a theoretical foundation for this work and covers essential concepts that are required for the following chapters. It is divided into two sections. First section explains concept and idea behind Internet of Things and provides necessary overview of IoT-Ticket to which this work is closely related. Second part covers theoretical background concerning cryptography and information security in relevant extend before the actual implementation can be discussed in more detail.

2.1 Internet of Things

The term “Internet of Things” was first used by Kevin Ashton in his RFID (Radio Frequency IDentification) related presentation in 1999 [1]. Since then interest towards it has grown gradually and during the 2000s it became recognized as an important research area [2, p. 12–24]. Today the Internet of Things (IoT) is considered a concept which could have major impact on the global economy in the future [3, p. 6–7].

As a concept, the IoT essentially includes the idea of connecting various physical objects (furniture, domestic appliances, cars, pets, factory floor equipment, plants, buildings etc.) around us to the Internet, where they can share resources related to themselves and interact with other networked entities. These physical objects are equipped with various embedded system technologies (computation, communication etc.) and have the ability to interact with their physical environment by measuring its phenomena with sensors and/or affecting to it with actuators. [4, p. 11–12] It is stated [5, p. 15] that the role of IoT is to bridge the gap between the physical world and its representation in information systems. Same study gives also the following more detailed definition for the Internet of Things:

“A world where physical objects are seamlessly integrated into the information network, and where the physical objects can become active participants in business processes. Services are available to interact with these ‘smart objects’ over the Internet, query their state and any information associated with them, taking into account security and privacy issues.” [5, p. 15]

The idea of connecting various physical objects to the Internet is not new. However, it has made topical by the confluence of several technology and market trends which enable to interconnect more and smaller devices cheaply and easily. These include: widespread adoption of IP-based networking, network connectivity becoming more pervasive, computing and communication technology becoming smaller, more efficient and less energy-intensive, advances in data analytics and rise of cloud computing. As such, IoT can be

seen as a concept which includes wide range of technologies which have been evolving for many decades. [6, p. 8]

The fundamental aspect of IoT is to collect and utilize large amounts of data that originates from various types of sensors embedded in different kind of physical objects. Having access to such data creates new possibilities for various applications. However, raw data in itself is not very useful. It must be refined and analyzed in order to really comprehend its meaning. This is essential since it establishes better possibilities to utilize collected data more meaningfully and productively in various IoT applications like in decision making and remote monitoring. As such, having capability to process (analyze, refine etc.) collected data is as equally important as collecting it. [7, p. 9–10]

Networking physical objects makes it also possible to control their surroundings if they are equipped with actuators. This, in turn, enables different kinds of control applications such as adjusting room lighting and heating remotely. Both, remote monitoring and controlling of physical objects can be seen as an essential part of today's IoT [8].

As a contrast, future visions of IoT contain ideas in which physical objects would have the ability to achieve common goals autonomously working as a group and solve problems requiring minimal human intervention [9, p. 6–7]. It is also envisioned that various physical objects in the future would have the ability to learn from refined information which is collected over a long period from different types of networked objects [7, p. 9–10]. In principle such abilities would allow various objects continuously optimize their operation and provide help to other objects if they are not able to perform their tasks alone.

2.1.1 General architecture and common elements

Based on the findings in [10] [11] [12] [13] it can be perceived that IoT systems have general architecture and set of common elements. These common elements include: physical objects (equipped with various technologies, sensors and actuators), network infrastructure (routers, switches, gateways etc.), cloud computing entities (scalable computing power and storage etc.) and various interfaces for end users and other systems. The elements can be further organized in architectural models as noted in most studies above. These models have variation in their accuracy but the core idea between them is generally consistent. To illustrate this, two models are briefly explored. First is a model according to IoT World Forum Architecture committee which consist of members such as Cisco, General Electric, IBM, Intel and Oracle [12, p. 3] is illustrated in Figure 1.

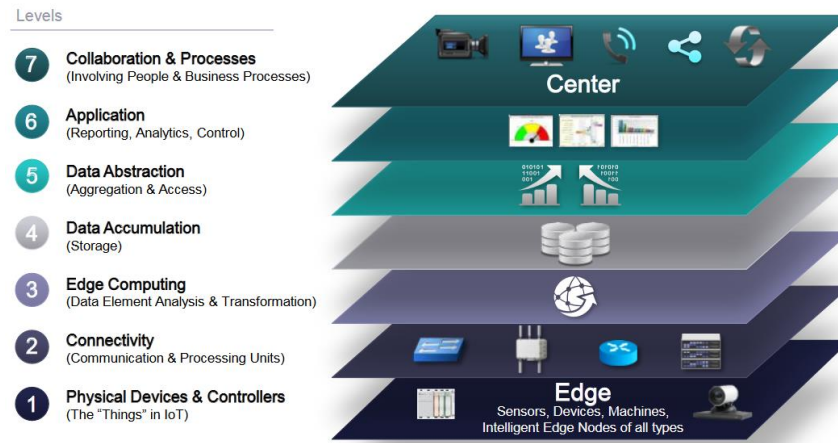


Figure 1. General architecture according to [12, p. 5].

This model consists of several layers. At the bottom layer are the physical objects which produce the actual data and relay it forward. Above these is the network infrastructure through which the data is transferred to cloud computing entities. Such entities process the data and store it in database from where it can be fetched by various types of applications (reporting, analytics etc.). These are located in the top section of the model.

A model according to [10] is illustrated in Figure 2. Compared to model above the idea is generally the same.

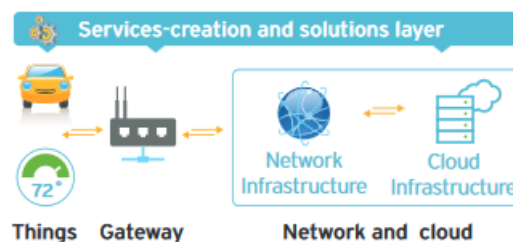


Figure 2. General architecture according to [10, p. 2].

On the left side in this model are physical objects. These communicate with gateways which provide them Internet connectivity. Next is the network infrastructure that is responsible for controlling the data flow between the physical objects and cloud infrastructure. Cloud infrastructure consists of large pools of virtualized servers and storage that are networked together. They process data collected from large number of physical objects and make it accessible for end users, various applications and services. [10, p. 1–2]

Based on these findings, general IoT architecture can be perceived as consisting of following elements: physical objects (equipped with sensors and/or actuators), communication infrastructure (routers, gateways etc.), cloud computing infrastructure (servers, storage etc.) and applications & services. These elements form a model where communication infrastructure links the physical devices and the cloud infrastructure together allowing

data to flow between every element, and where applications & services are built on top of cloud infrastructure.

2.1.2 Wapice IoT-Ticket

IoT-Ticket [14] is an IoT platform developed by Wapice Ltd. It is introduced here due to its close relation to this work but also to briefly illustrate main characteristics and appearance of commercial IoT product.

The most visible part of IoT-Ticket is its web based user interface which allows users to monitor and control connected devices, analyze and visualize collected data and generate reports [15]. Monitoring and control takes place under various dashboards which consist of various widgets (e.g. meters, displays, buttons etc.) that allow users to interact with their devices. Interface Designer, which is used to create various dashboards, is illustrated in Figure 3. Data analytics and reporting can be done using similar web based user interface.

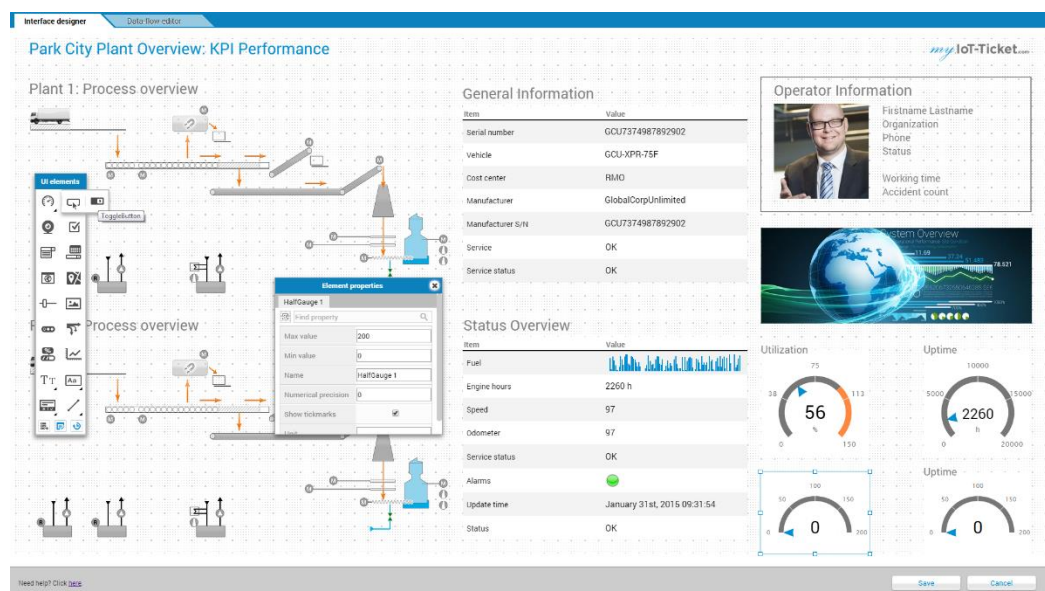


Figure 3. Interface Designer [16].

IoT-Ticket has support for various communication interfaces that allow different types of devices to be connected to the system. Such interfaces include REST (Representational State Transfer) and OPC UA (Object Linking and Embedding for Process Control Unified Architecture). In addition to these, connectivity of the system can be enhanced through specific proprietary hardware which supports e.g. digital I/O (Input/Output), Ethernet, CAN (Controller Area Network), RS-232 (Recommended Standard 232), RS-485 (Recommended Standard 485) and 1-Wire. More devices and protocols can be supported through various adapter/gateway implementations which utilize previously mentioned

REST API (Application Programming Interface). Existing implementations of this include OPC (Object Linking and Embedding for Process Control) and MQTT. [15] Overview of supported interfaces is illustrated Figure 4.

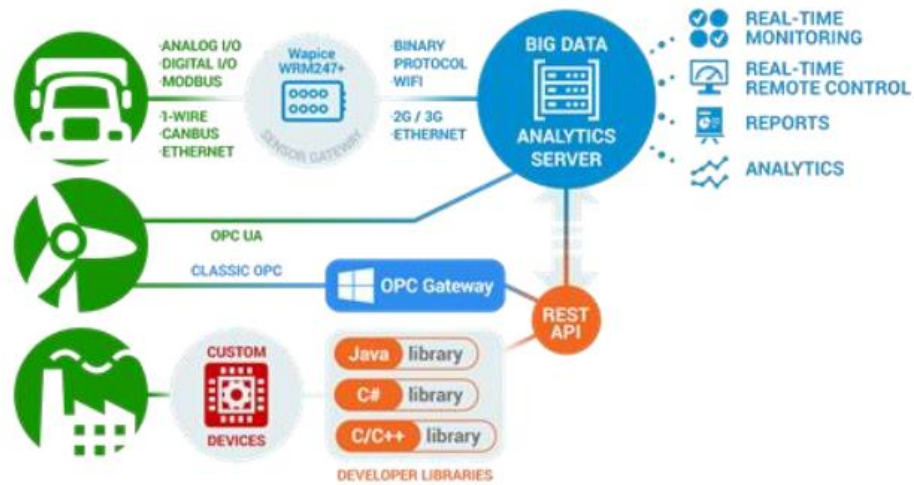


Figure 4. Overview of the supported communication interfaces, modified from source [17, p. 1].

In terms of the actual implementation done in this work it is essential to briefly cover the data model behind IoT-Ticket when its REST API is used convey data into the system. The model is used to represent various data sources in unified manner. It is hierarchical and consists of three main elements. These are:

- enterprise
 - device and
 - data node.

An enterprise is a root element. It can contain multiple devices and a device in turn can have multiple data nodes. Data nodes represent the actual data of particular device in the system. [17, p. 2–3] These elements contain also additional information. In terms of device element this includes [17, p. 6]:

- name (required),
- manufacturer (required),
- type,
- description and
- attributes (key value pairs).

In terms of data node element this includes [17, p. 15–17]:

- name (required),
- path (e.g. core/temperature),
- value (required),

- timestamp,
- unit and
- data type.

Some of the information is required and some is optional. This is denoted in parenthesis.

2.2 Security fundamentals

This chapter presents essential security related concepts before details of the actual implementation are explained in more detail. Here concepts such as symmetric and asymmetric cryptography, and message authentication are discussed. In addition to these, specific algorithms and practices which are used to implement desired features in practice in this work are also illustrated.

2.2.1 Symmetric and asymmetric cryptography

Encryption is a process in which plaintext message is transformed into cipher text using encryption algorithm and secret key. Decryption is a process in which cipher text is transformed into plaintext using decryption algorithm and secret key. Plaintext refers to original message and cipher text to coded message. [18, p. 30] Symmetric cryptography refers to cryptographic system where encryption and decryption algorithms use the same key. Asymmetric cryptography refers to cryptographic system where encryption and decryption algorithms use different keys.

Symmetric encryption, also referred as a conventional encryption or single-key encryption, was the only type of encryption before the invention of asymmetric cryptography in 1970 [18, p. 30]. Symmetric encryption schemes (ciphers) can be classified into two categories based on their operation: block ciphers and stream ciphers. Block ciphers take block of plaintext elements to their input and produce equal length block of encrypted elements to their output. Stream ciphers in turn take stream of bits or bytes to their input and produce corresponding stream of encrypted bits or bytes to their output. [18, p. 64] Ciphers in both categories are based on different substitution and permutation schemes in order to produce cipher text counterpart from plaintext input [18, p. 36]. Various implementations can be found from these two categories. In case of block ciphers these include e.g. AES (Advanced Encryption Standard) [18, p. 135] and DES (Data Encryption Standard) [18, p. 62]. In case of stream ciphers these include e.g. RC4 (Rivest Cipher 4) [18, p. 189]. Figure 5 illustrates conventional symmetric cryptosystem.

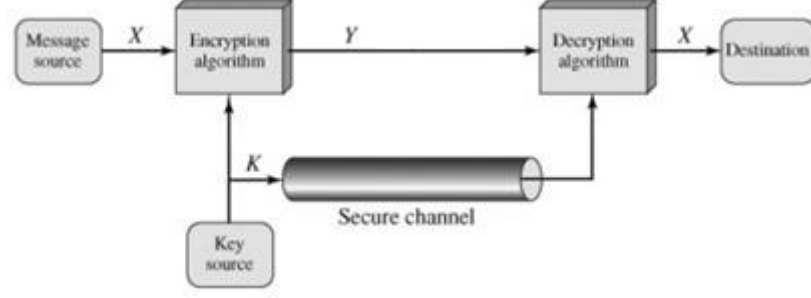


Figure 5. Conventional symmetric cryptosystem, modified from source [18, p. 31].

On the left side is plaintext message (X) which is desired to be sent to destination so that any intermediate observer cannot resolve its meaning. Plaintext message (X) and key (K) (generated at sender side and distributed in advance through secure channel to destination) are input to symmetric encryption algorithm which is used to transform plaintext message into cipher text (Y) which is then sent to destination. At receiver side cipher text (Y) and key (K) are input to symmetric decryption algorithm which is used to transform cipher text (Y) back into original plaintext (X). Mathematically this can be expressed with two equations. In case of encryption:

$$Y = E(K, X), \quad (1)$$

where Y is cipher text, E is encryption function, K is shared key and X is plaintext. Decryption in turn can be expressed as follows:

$$X = D(K, Y), \quad (2)$$

where X is plaintext, D is decryption function, K is shared key and Y is cipher text. [18, p. 31–32]

In contrast to symmetric encryption, asymmetric encryption, also called public-key cryptography, uses key pair instead of single shared key. It is also profoundly different from theoretical point of view compared to symmetric cryptography since it is based on mathematical functions, specifically one-way functions [18, p. 267], rather than on various substitution and permutation schemes. [18, p. 258]

Asymmetric cryptography uses one key for encryption and another related key for decryption. These keys form a pair in which other key is called public key and another private key. The private key, as the name suggest, is kept secret (private) and it is not distributed to others, unlike the public key counterpart. Plaintext messages are encrypted with public key and can be decrypted using only corresponding private key. [18, p. 260–262] In addition to encryption/decryption, asymmetric cryptography can be used also in authentication. In such case, plaintext message is encrypted using sender's private key (not receiver's public key as in encryption/decryption scheme). When such message is received, it can be transformed to plaintext only by using sender's public key. As such, it

can be verified that message originates from sender which owns private key counterpart. [18, p. 264] Various implementations can be found relying on asymmetric cryptography such as RSA (Rivest-Shamir-Adleman) and Elliptic Curve [18, p. 266]. Figure 6 illustrates conventional asymmetric cryptosystem when it is used to encrypt/decrypt messages between two parties.

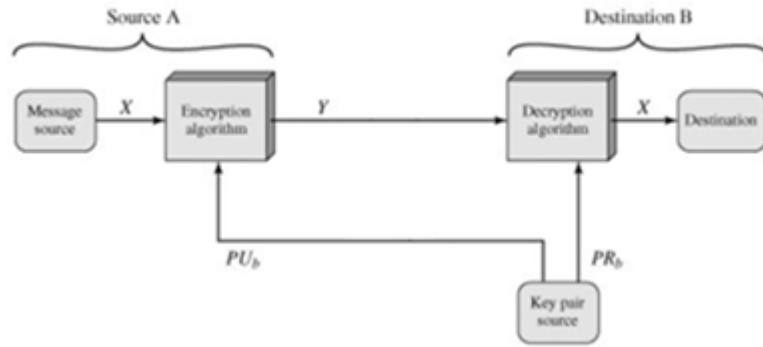


Figure 6. Conventional asymmetric cryptosystem, modified from source [18, p. 263].

On the left side (A) is plaintext message (X) which is desired to be sent to destination (B) so that any intermediate observer cannot resolve its meaning. Plaintext message (X) and receiver's public key (PU_b) (generated by B and publicly distributed in advance to A) are input to asymmetric encryption algorithm which is used to transform plaintext message into cipher text (Y) which is then sent to destination (B). At receiver side (B) cipher text (Y) and receiver's private key (PR_b) are input to asymmetric decryption algorithm which is used to transform cipher text (Y) back into original plaintext (X). Mathematically this can be expressed with two equations. In case of encryption:

$$Y = E(PU_b, X), \quad (3)$$

where Y is cipher text, E is encryption function, PU_b is B's public key and X is plaintext. Decryption in turn can be expressed as follows:

$$X = D(PR_b, Y), \quad (4)$$

where X is plaintext, D is decryption function, PR_b is B's private key and Y is cipher text. [18, p. 263]

2.2.2 RC4 stream cipher

RC4 (Rivest Cipher 4) is a variable key-size stream cipher with byte-oriented operations. It was designed in 1987 by Ron Rivest for RSA Security and was trade secret until it was anonymously posted on the Internet in September 1994. RC4 is perhaps the most popular symmetric stream cipher and has applications in various wireless network protocols and in TLS (Transport Layer Security). [18, pp. 189,191] Before explaining the algorithm and

its operation more specifically, some fundamental details are first covered regarding stream ciphers in general.

A typical stream cipher takes stream of plaintext bytes to its input and produces corresponding stream of encrypted bytes to its output. Encryption key, which is provided for the cipher, is used to generate stream of pseudorandom bytes. These pseudorandom bytes, also referred as key stream, are used to perform the actual encryption where plaintext input is transformed to cipher text output. Encryption happens one byte at the time and uses bitwise XOR (exclusive OR) operation. Encryption process is illustrated in Figure 7. Here bitwise XOR operation is abbreviated by \oplus . [18, p. 190]

$$\begin{array}{rcl}
 11001100 & \text{plaintext} & \\
 \oplus 01101100 & \text{key stream} & \\
 \hline
 10100000 & \text{ciphertext} &
 \end{array}$$

Figure 7. Encryption using XOR operation [18, p. 190].

Here a byte is taken from plaintext input and it is combined with corresponding byte from key stream using XOR operation to form a cipher text byte. Process is repeated by sifting bytes from both streams side by side as long as there are bytes in the input byte stream. Decryption is done similarly at the receiver side using identical pseudorandom byte stream (generated using the same shared encryption key). In this case bitwise XOR operation is applied between pseudorandom byte and cipher text byte streams to produce the original plaintext byte stream. Decryption of cipher text byte is illustrated in Figure 8. [18, p. 190]

$$\begin{array}{rcl}
 10100000 & \text{ciphertext} & \\
 \oplus 01101100 & \text{key stream} & \\
 \hline
 11001100 & \text{plaintext} &
 \end{array}$$

Figure 8. Decryption using XOR operation [18, p. 190].

With RC4, encryption and decryption are similar to above. However, characteristics that are specific for RC4, are the details on how the pseudorandom key stream is generated. The key stream generation process in RC4 consists from two initialization phases and actual stream generation phase. In the first initialization phase two 256 byte arrays, namely S (state) and T (temporary), are created. S is initialized with values from 0 to 255 in ascending order. T is initialized with provided key in such way that it fills the whole array (valid key length is therefore from 1 to 256 bytes). Array K contains the actual key. Pseudocode for the first initialization phase is illustrated below. [18, p. 191–192]

```

/*
Initialization

```

```

*/
for i = 0 to 255 do
S[i] = i;
T[i] = K[i mod keylen];

```

In the second initialization phase T is used to produce initial permutation of S . This is illustrated in the pseudocode below. *Swap* denotes of changing places of two elements in the S array. [18, p. 192]

```

/*
Initial Permutation of S
*/
j = 0;
for i = 0 to 255 do
j = (j + S[i] + T[i]) mod 256;
Swap(S[i], S[j]);

```

In the next phase the actual key stream is generated which is then used to encrypt or decrypt bytes using XOR operation. Here array S is iteratively cycled through simultaneously swapping places of two elements in it. In the end of each cycle pseudo number byte is calculated (denoted by k) which is then used to encrypt or decrypt corresponding byte in plaintext or cipher text stream. Pseudocode example from key stream generation is illustrated below. [18, p. 192]

```

/*
Stream Generation
*/
i, j = 0;
while(true)
i = (i + 1) mod 256;
j = (j + S[i]) mod 256;
Swap(S[i], S[j]);
t = (S[i] + S[j]) mod 256;
k = S[t];

```

RC4 is relatively simple to implement. It is also computationally fast compared to other symmetric ciphers such as DES (Data Encryption Standard) and 3DES (Triple Data Encryption Standard). [18, p. 191] Although RC4 has its good properties, it has also flaws and weaknesses that have the possibility to predispose it to key recovery attacks [19, p. 89]. Therefore, use of other symmetric ciphers, such as AES is recommended [20].

2.2.3 Message authentication

The message authentication (also referred to data-origin authentication) refers to scheme which allows to verify the origin of the message and to detect if the message sent over the unsecured channel is tampered/distorted between the sender and the receiver. It is important to realize that message authentication does not address data confidentiality. [21, p. 155] Data confidentiality refers to encrypting the message so that it cannot be directly comprehended by unwanted parties.

Message authentication is associated with a method of calculating small piece of information from a message and transmitting it along with the original message. Specifically, such piece of information is called as MAC (Message Authentication Code) which is a sort of a checksum (under certain assumptions) calculated from the message. It is calculated using MAC algorithm and secret key which is shared between sender and receiver of the message. Thus, only sender and receiver are able to calculate valid MAC. If the message is tampered during the transmission it will be noticed by the receiver since received MAC does not match with the one which is calculated by receiver itself. This allows only the sender and receiver (parties that have access to secret key) calculate valid MAC and hence detect tampering and be sure that the message is originated from expected source. [18, pp. 318, 320] The process of calculating MAC, merging it with message which and sending these over unsecured channel is illustrated in Figure 9.

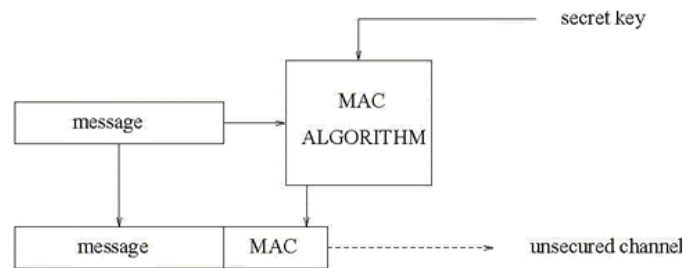


Figure 9. Message Authentication Code calculation process [22].

In general, there are two common methods to derive MACs which have properties that allow recipient to verify the origin of the message and to detect if message has been tampered during the transmission. One of these methods is HMAC (Hash-based Message Authentication Code) which utilizes hash functions to produce MAC. Hash functions are algorithms that take arbitrary length data to their input and produce fixed length data to their output. In addition to HMAC, another common method to calculate MAC is CMAC (Cipher-based Message Authentication Code). Instead of hash functions, CMAC is based (as the name suggest) on block ciphers. [18, p. 352] [23, p. 7–10]

Using MACs, message can be authenticated and its integrity guaranteed. However, usually the requirement is also to achieve confidentiality (content encryption). Figure 10 illustrates one possible practice which provides both confidentiality and message authentication. It uses hash function together with symmetric cipher.

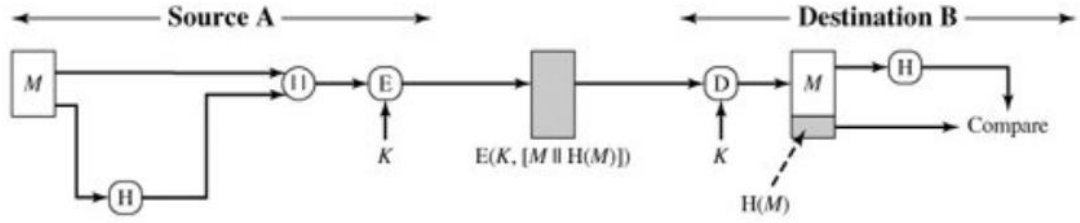


Figure 10. Using hash function with symmetric cipher, modified from source [18, p. 330].

Here message M is sent from A to B . First, hash is calculated from message M using hash function H . Next the result (hash) is merged (denoted by $||$) with message M which are then encrypted together using symmetric encryption algorithm E and shared key K . Encrypted message $E(K, [M || H(M)])$ is then sent to destination. At the destination message is first decrypted using symmetric decryption algorithm D and shared key K . Hash $H(M)$ is then extracted from the decrypted message M and compared to new hash which is calculated from M using same hash function H . If the hashes match, received message is considered valid (not tampered or distorted during transmission) and originates from authentic sender.

2.2.4 SHA-256 algorithm

In general, hash algorithms are functions that take arbitrary length message to their input and produce fixed length message (hash) to their output which represents given input. Length and content of the output depends on hash algorithm. Set of well-known and widely used hash algorithms can be recognized from SHA (Secure Hash Algorithm) prefix in their name. They are developed by the National Institute of Standards and Technology (NIST) and are published under FIPS (Federal Information Processing Standards) 180-4 standard. The standard defines five variants of the SHA algorithm: SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512. Most noticeable difference between these variants is the length of the hash they produce to their output. In case of SHA-1 length is 160 bits. For SHA-224, SHA-256, SHA-384 and SHA-512 length is 224, 256, 384 and 512 bits respectively. [18, p. 253] [24, p. 3] Detailed description of SHA-256 variant is given below.

SHA-256 algorithm utilizes six logical functions and total of sixty-four constants in the actual hash calculation process. Functions take 32-bit words (x, y, z) as their input and produce equal length (32-bit) output. They are defined as follows [24, pp. 4–6, 10]:

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z) \quad (5)$$

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z), \quad (6)$$

$$\Sigma_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x), \quad (7)$$

Before hash calculation starts, hash value is initialized. Hash value consists of eight 32-bit words that are denoted by $H_0^{(i)}, H_1^{(i)}, \dots, H_7^{(i)}$. They are initialized as follows using hexadecimal representation [24, p. 15]:

$$\begin{aligned} H_0^{(0)} &= 6a09e667, & H_1^{(0)} &= bb67ae85, \\ H_2^{(0)} &= 3c6ef372, & H_3^{(0)} &= a54ff53a, \\ H_4^{(0)} &= 510e527f, & H_5^{(0)} &= 9b05688c, \\ H_6^{(0)} &= 1f83d9ab, & H_7^{(0)} &= 5be0cd19. \end{aligned}$$

When all the preprocessing is done, algorithm starts iterating through message blocks $M_t^{(i)}$ and calculating actual hash. For each iteration ($1 \leq i \leq N$) following operations are performed.

Below, W_t (message schedule) is used as a workaround together with variables $a, b, c, d, e, f, g, h, T_1$ and T_2 to convey data between operations. W_t consists of sixty-four 32-bit words which are denoted by W_0, W_1, \dots, W_{63} . Addition (+) is performed modulo 2^{32} . [24, p. 4–5]

1.

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \leq t \leq 63 \end{cases}$$

2.

$$\begin{aligned} a &= H_0^{(i-1)} & b &= H_1^{(i-1)} & c &= H_2^{(i-1)} & d &= H_3^{(i-1)} \\ e &= H_4^{(i-1)} & f &= H_5^{(i-1)} & g &= H_6^{(i-1)} & h &= H_7^{(i-1)} \end{aligned}$$

3.

for $t = 0$ to 63 do:

$$T_1 = h + \Sigma_1^{\{256\}}(e) + Ch(e, f, g) + K_t^{\{256\}} + W_t$$

$$T_2 = \Sigma_0^{\{256\}}(a) + Maj(a, b, c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2$$

4.

$$H_0^{(i)} = a + H_0^{(i-1)} \quad H_1^{(i)} = b + H_1^{(i-1)} \quad H_2^{(i)} = c + H_2^{(i-1)}$$

$$H_3^{(i)} = d + H_3^{(i-1)} \quad H_4^{(i)} = e + H_4^{(i-1)} \quad H_5^{(i)} = f + H_5^{(i-1)}$$

$$H_6^{(i)} = g + H_6^{(i-1)} \quad H_7^{(i)} = h + H_7^{(i-1)}$$

After performing all the steps (1–4) illustrated above total of N times, hash calculation is finished. The result is then obtained as follows [24, p. 21–22]:

$$H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)} \parallel H_6^{(N)} \parallel H_7^{(N)}.$$

Here eight 32-bit words are merged (denoted by \parallel) together to form sequence of bits which represents complete 256-bit hash. This bit sequence is the final output from the SHA-256 algorithm.

2.2.5 Overview of TLS and X.509 certificates

TLS (Transport Layer Security) is a cryptographic protocol designed to provide secure communication between client and server over insecure communication infrastructure [25, p. 1]. It operates on top of TCP (Transmission Control Protocol) providing secure transport for upper layer protocols such as HTTP (Hypertext Transport Protocol) [25, p. 2–3]. The development of the protocol started at Netscape and was originally called SSL (Secure Sockets Layer) rather than TLS. Name was changed to TLS when the protocol was migrated from Netscape to IETF (Internet Engineering Task Force). First TLS named version (1.0) came out in 1999. Current version (1.2) was released in 2008. [25, p. 3–4]

TLS is implemented via so called record protocol. It is a container for four sub protocols (change cipher spec protocol, alert protocol, handshake protocol and application data protocol) which are carried inside the record protocol. [25, p. 24–25] In this chapter, main features of TLS are illustrated through the handshake protocol.

The handshake protocol is used to negotiate connection parameters, validate certificates and agree shared master secret (used to encrypt/decrypt actual application data) between client and server [25, p. 26]. Figure 12 illustrates the message sequence in handshake protocol which is also known as mutual handshake. In mutual handshake, both client and server authenticate each other [25, p. 32–33]. In contrast, when only the server authenticates, procedure is called full handshake [25, p. 26].

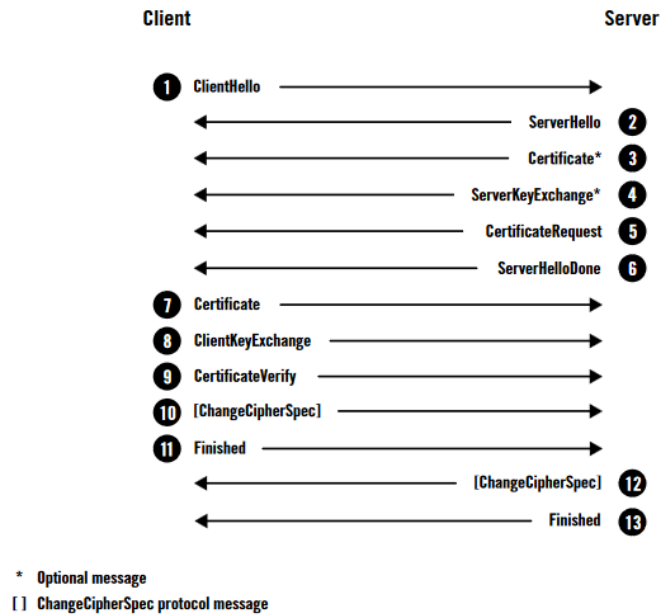


Figure 12. Mutual handshake in TLS [25, p. 33].

Every time the client initiates a new TLS session it sends *ClientHello* message to server. This message contains best protocol version supported by the client, list of supported cipher suites, random data (used as a part of the master secret calculation process), list of compression methods (usually not used in TLS) and extensions. With *ClientHello* message client proposes connection parameters to server from which server selects the ones that are eventually used. [25, p. 28–29]

The server responds with *ServerHello* message. It contains parameters selected for the TLS session along with server’s random data (used as a part of the master secret calculation process). A handshake sequence continues with *Certificate* message. It is typically responsible for carrying server’s X.509 certificate chain (set of X.509 certificates) which is used to authenticate the server. Next is *ServerKeyExchange* message which is used to transfer server’s key exchange parameters to client. These parameters are used to generate a value called premaster secret which is used to derive another value called master secret. Master secret is used to encrypt and decrypt the actual application data sent between client and server in application data sub protocol. *CertificateRequest* is used to request authentication from client. It contains lists of acceptable client certificates. *ServerHelloDone* ends server side handshake and signals client to continue. [25, p. 29–31]

The client continues mutual handshake with *Certificate* message by sending its certificate to server. *ClientKeyExchange* message is used to transfer client’s key exchange parameters to server. It is client’s contribution to key exchange and premaster secret generation process. A subsequent message *CertificateVerify* is used to prove that the client has the private key related to public key embedded in the certificate which was sent along with the *Certificate* message. For this *CertificateVerify* contains signature of all the handshake

messages exchanged until this point produced with the client's private key. *ChangeCipherSpec* message is used by the client to signal that it has all information which it needs to generate master secret and now switches to encryption. It is followed by *Finished* message which is the first message encrypted using master secret. It contains MAC calculated from all sent and received handshake messages. After the client has finished its handshake sequence server also sends corresponding *ChangeCipherSpec* and *Finished* messages. If there were no errors and received MACs were correct, TLS session is established and application data can be exchanged between client and server using commonly agreed master secret. [25, p. 30–34]

During the TLS handshake client and server agree on cipher suite. It is essentially a set of key exchange, authentication, encryption/decryption and MAC algorithms which are used to exchange data securely between client and server [25, p. 50]. Figure 13 illustrates different components which constitute a cipher suite. Here key exchange algorithm is ECDHE (Ephemeral Elliptic curve Diffie-Hellman) and authentication algorithm is RSA. Application data will be encrypted/decrypted using symmetric AES cipher in GCM (Galois/Counter Mode) mode with 128-bit key. MAC algorithm is SHA-256. TLS has support for various symmetric block and stream ciphers such as 3DES, AES and RC4 [25, p. 42].

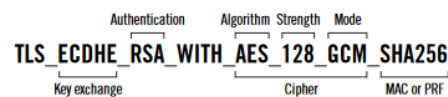


Figure 13. Example of cipher suite used in TLS [25, p. 51].

Key exchange algorithms are used to produce a value called premaster secret which is used to calculate yet another value called master secret. The master secret is the secret key which is used to encrypt and decrypt the application data between client and server during TLS session. RSA is effectively the standard key exchange algorithm; client simply generates premaster secret, encrypts it with server's public key (included in server certificate) and sends the result to server which decrypts it with its private key. After this both sides have the same premaster secret which they use to generate master secret. However, RSA has the weakness that all encrypted data can be decrypted if server's private key is compromised. Because of this, following key exchange algorithms should be used: DHE (Ephemeral Diffie-Hellman) and ECDHE. They have property called forward secrecy. [25, p. 35–41] Forward secrecy prevents decrypting messages from previous sessions if server's private key is compromised [25, p. 257].

Authentication during the handshake is most commonly performed using RSA algorithm in conjunction with the X.509 certificates. Certificates are used to enable secure communications between parties which have never met before. The process is based on trusted third parties called CAs (Certificate Authorities) which issue certificates for untrusted parties. These certificates are essentially digital documents which contain information

about the certificate itself, its issuer and its subject. The subject is the end entity for which this certificate is assigned. Subject has the private key which public key counterpart is embedded into certificate among other information. Issuer is the authority who has issued the certificate. Issuer's private key is used to sign the certificate. X.509 is a standard that defines the format for certificates which are commonly used on the Internet. [25, p. 66–68] [26]

Certificates are signed with issuer's private key and can be therefore validated using corresponding public key. This public key is embedded in so called CA certificate which is distributed for end entities before the actual communication takes place with some out of band process. Later when an end entity receives certificate from some unknown source, it uses the public key embedded in CA certificate to check validity of the received certificate. This is essentially the process through which authenticity of unknown source is validated with certificates. Moreover, certificates can be chained, meaning that certificates are signed by intermediate authorities. In such case, receiver must verify all the intermediate certificates until root certificate in order to check the validity of end entity. [26] Certificate chains are used for security, technical and administrative reasons [25, p. 71].

3. MQTT PROTOCOL

This chapter describes core functionality and characteristics of the MQTT protocol. In addition to its core features, structure of relevant MQTT control packets is also illustrated.

3.1 Overview

MQTT (Message Queuing Telemetry Transport) is an application level protocol which operates on top of the TCP/IP (Transmission Control Protocol/Internet Protocol) stack. It is simple, lightweight and easy to implement protocol which utilizes client-server publish-subscribe messaging pattern. The official MQTT standard states that the characteristics of the protocol make it ideal to use in contexts like M2M (Machine to Machine) or IoT where a small code footprint is required and/or network bandwidth is at a premium. [27, p. 1–2]

MQTT was originally developed at IBM in 1999 where it was designed to be lightweight, bandwidth efficient, simple to implement, agnostic about delivered data, aware of the session and able to provide QoS (Quality of Service) for delivered data. During its early days MQTT was used internally at IBM for proprietary embedded systems. However, in 2010 IBM decided to release the protocol royalty free for everyone to use. At the time the protocol carried version number 3.1. A few years later MQTT was moved under OASIS (Organization for the Advancement of Structured Information Standards) and in 2014 it was released under open OASIS standard with version number 3.1.1. Version 3.1.1 is also the newest version of the protocol. [28]

3.2 Publish-subscribe pattern

A system based on publish-subscribe pattern relies on a central node, known as message broker (server), to which communicating end points (clients) are connected. From client's perspective all messages in such system are sent to and received from broker. Thus, clients are not aware of each other and all they see is the connection with the broker. The broker has the responsibility to receive messages from clients and to deliver them to appropriate recipients. In MQTT, broker establishes its delivering logic on so called topics; when client sends a message, it assigns the message with some topic. Moreover, each client has indicated one or more topics to broker which they are interested in. By having this information, the broker can then look the topic in the received message and deliver it to appropriate recipients which are interested in this topic. [29]

In MQTT when client indicates its interest about certain topic it is said that client subscribes messages from that certain topic and when client sends a message assigned with certain topic it is said that client publishes message to that certain topic. As such, when

client has message to send, it publishes it to the topic and based on subscriptions the broker delivers the message to clients which have subscribed the topic. [29]

3.3 Main features and functionality

MQTT is based on publish-subscribe pattern where messages are delivered between clients via central message broker. Although the protocol is considered simple and easy to implement there are still number features which might not be self-evident and therefore require closer exploration in order to understand how the protocol actually works. The key features and functionality of the protocol are covered next.

3.3.1 Connection establishment

When a client establishes connection with the broker, it sends CONNECT packet to broker. With CONNECT packet client configures set of parameters that are used for the connection with the broker. These parameters control e.g. what happens if client disconnects ungracefully from broker, or whether particular messages should be stored for the client if it goes offline. The most essential ones are described below.

- Client identifier
 - Client identifier identifies the client for the broker. The broker will use it to keep track of the state of the client. It is unique for each client. In MQTT 3.1.1 it is also possible to set client identifier empty. In such case connection has no state and the client is not allowed to establish a persistent session with the broker. [30]
- Clean session
 - With clean session flag client indicates whether it wants to establish a clean or persistent session with the broker. If clean session is requested (flag set to true), broker will not restore nor start storing any state for the client and will purge all information from previous persistent session. However, if flag is set to false (indicating persistent session), broker will restore previous session (if any) for the client. This means that any topic subscriptions made by client in previous session are restored and messages which client had subscribed (with QoS 1 or 2) and which were received when client was offline are sent to it. If persistent session is requested, broker starts storing state for the current session. [30]
- User name and password
 - User name and password are used for authentication and authorization to broker. They are sent in plaintext.
- Last will
 - Last will is a part of last will and testament feature of the MQTT. It is used to notify other clients if a client disconnects ungracefully from the

broker. In case of such event, the broker, on behalf of disconnected client, sends predefined message to predefined topic. Both the message and the topic are defined by the disconnected client during the connection establishment. [30]

- Keep alive
 - Keep alive is used as a time interval which is allowed to elapse between consecutive messages sent from client to broker. In case there are no messages to be sent, client sends PING request to broker to which broker must respond with PING response. This mechanism is used to determine whether either side of the communication is still reachable. [30] [27, p. 27]

3.3.2 Publish and subscribe messages, topics

When MQTT client publishes application data to specific topic it essentially sends PUBLISH packet to broker. PUBLISH packet contains actual application data and topic but also other important information such as retained flag and QoS level. [31]

Retained flag is indication to broker whether it must retain received message for the topic or not. If topic contains retained message and some client subscribes it, broker immediately sends retained message to this client. Retained message can be considered as last known good value saved for topic (only latest message for topic is kept). In terms of QoS level, client can choose between three QoS levels when it publishes the message: 0 (at most once), 1 (at least once) and 2 (exactly once). QoS levels are used to provide different guarantees for message delivery between client and broker. [31] They are discussed later in more detail.

When message is published to topic, broker checks the topic and delivers the message to clients that are subscribed corresponding topic. Messages that are sent from broker to client are also sent as PUBLISH packets. However, these packets are not direct copies of the received ones although they have same content in their payload portion (holds the actual application data). For example, the QoS level might change during message delivery which affects the fields in the PUBLISH packet. [31]

When MQTT client subscribes topic it sends SUBSCRIBE packet to broker. Such packet contains essentially a list of topics with QoS levels with which client wants to subscribe corresponding topic. Client can specify the QoS levels independently of topics and other clients. [31]

In MQTT, topics have hierarchical levels where the levels are separated from each other using forward slash (/). This is similar to URLs (Uniform Resource Locator) used in the Internet. Moreover, topics can be subscribed one topic at the time or by using so called wildcards which can represent simultaneously multiple topics. In MQTT there are two

types of wildcards: single- and multilevel wildcards. Single level wildcards are denoted with plus sign (+) and multilevel wildcards with octothorpe (#). Here are couple of examples of their use. In case of single level wildcard, following topic: *level1/+/level3* is equivalent e.g. with *level1/x/level3* and *level1/y/level3*. In case of multilevel wildcard, following topic: *level1/level2/#* is equivalent e.g. with *level1/level2/x/y* and *level1/level2/y/x*. [32]

MQTT is data agnostic protocol. This means that underlying application data embedded in payload portion of PUBLISH packet can be anything from binary representation to high level format such as XML (Extensible Markup Language), JSON (JavaScript Object Notation) or CSV (Comma-Separated Values). The broker, which delivers messages between clients, is able to deliver the message without having the ability to interpret the actual application data in payload portion. Handling the payload is responsibility of clients. This also means that payload portion can be encrypted so that intermediate broker or any unwanted client cannot deduce its contents without knowing secret key. [31]

3.3.3 Quality of service

In MQTT, quality of service (QoS) is essentially an agreement between sender and receiver in which they agree the assurance of message delivery concerning messages that are sent from client to broker and vice versa. QoS agreement is client and topic specific: when client publishes messages to topic it sets QoS level independently for each message and when client subscribes messages it independently sets QoS level for each topic. QoS level can therefore get downgraded. This happens if one client publishes with higher QoS than another has subscribed same topic. As mentioned earlier there are three distinct QoS levels in MQTT: 0 (at most once), 1 (at least once) and 2 (exactly once). [33]

With QoS level 0, which is also the simplest of the levels, sender just sends PUBLISH packet without waiting any confirmation or acknowledgement from the receiver to it. In such case PUBLISH packet is received at most once. [33]

In QoS level 1 every PUBLISH packet is received at least once. When QoS level 1 is used, every PUBLISH message is required to be acknowledged by the receiver with PUBACK packet. If the acknowledgement is not received in suitable window of time, PUBLISH packet is sent again which might happen several times. When QoS is equal or higher than 1, all transmitted PUBLISH packets are identified with packet identifier field (holds value greater than zero). Using the field receiver knows what packet it acknowledges with PUBACK packet. With QoS 0 packet identifier field is set to 0. [33]

QoS level 2 ensures that receiver processes every PUBLISH packet exactly once. In QoS 2, minimum of four packets are sent between sender and receiver. First packet in the sequence is the actual PUBLISH packet. When such packet is received, receiver is responsible of storing reference to the packet using its packet identifier. This is to prevent

the receiver from processing same PUBLISH packet twice. PUBLISH packet is then acknowledged by the receiver with PUBREC packet (contains original packet identifier). After receiving PUBREC packet sender can discard the original PUBLISH packet knowing that receiver has successfully received it. When PUBREC is received, sender stores reference to this packet and responds with PUBREL packet (contains also original packet identifier). After receiving PUBREL the packet, receiver can discard every state related to the packet identifier and respond with PUBCOMP packet. PUBCOMP is final packet and ends QoS level 2 packet delivery. If assumed packet is not received in suitable time window in any of the above delivery stages, previous packet is always sent again. [33]

If the client has made subscriptions with QoS level 1 or 2 and requested persistent session, broker will store packets for the client until it confirms them received. This is also the case if client goes offline: broker will store/queue packets until client reconnects with persistent session and confirms packets received. If client reconnects with clean session all previously stored packets are discarded. Packets that are subscribed with QoS level 0 are not stored. [34]

3.3.4 Security

MQTT itself does not provide versatile features regarding security: protocol concerns mainly authentication and authorization using client identifier, username and password. As such, protocol does not explicitly specify e.g. how application data should be encrypted or its integrity checked when carried within PUBLISH packets. Although MQTT protocol might not have diverse features regarding security, there are various ways to incorporate them. E.g. TLS can be used as underlying protocol instead of plaintext TCP to provide authentication, encryption and integrity check. Such features can be also implemented on application level: applications themselves can specify arbitrary data format making it possible to implement encryption and data integrity check in various formats inside PUBLISH packet's payload portion. [35] [36]

Client identifier, username and password are sent within CONNECT packet when connection is established with broker. When credentials are used, broker authenticates client and authorizes what resources (essentially topics) it is allowed to use. This of course requires that broker must be configured properly before the clients start connecting to it. It is also important to notice that all this information (client identifier, username and password) is sent in plaintext within CONNECT packet and is therefore visible to any intermediate network equipment if transferred on top of plaintext TCP connection. Because of this, CONNECT packet (if credentials are used) should be always sent on top of protocol which provides confidentiality for transferred data. [36]

When TLS is used as underlying protocol all MQTT packets can be encrypted and their integrity checked. In addition, clients can authenticate themselves to broker and broker can authenticate itself to the clients using X.509 certificates. As a contrast to plaintext

username and password, X.509 certificates provide better method to authenticate involved parties. However, certificates must be generated and provisioned along with private keys which makes them generally more challenging to manage. [37] [38]

If MQTT is used on top of TLS, TCP port 8883 is used on broker side [39]. It is standardized for secure MQTT connections. If MQTT is used over plaintext TCP connection port 1883 is used instead [40]. It is worth noting that TCP and TLS connection can be mixed and not all clients have to connect in same manner to broker: one client can connect with TLS and other with plain text TCP. In such case the data is not encrypted the entire way from one client to another.

TLS provides set of good features but on the other hand is computationally expensive and complex protocol. In addition, there might be also some situations where target platform simply has no support for it. In such situations, it might be the case that TLS just cannot be used. However, with MQTT it is possible to implement security features also on application layer.

When security features are taken into use on application layer, they are in practice implemented into PUBLISH packet's payload portion where the actual application data resides. As already noted earlier, payload can have arbitrary data format and be e.g. encrypted (MQTT is data agnostic protocol). When encryption is applied to payload, it stays encrypted all the way from source to destination (end-to-end (E2E) encryption) and only specific clients are able to recover its contents. This provides confidentiality for transferred data but also a method for authentication: only the clients which hold the secret key are considered authentic and consequently only they can comprehend the contents in the payload. However, payload's integrity cannot be guaranteed when it is just encrypted: any intermediate node e.g. malicious broker could modify the payload without recipients being able to notice it. To prevent this, MAC can be calculated from payload and incorporated into it before encryption. This ensures that to be able to modify any part of the payload, intermediate node would have to know the secret key. Using such primitives, desired properties for security can be implemented on application level. This is particularly useful if TLS cannot be used for some reason. [41] [42]

3.4 Control packets

MQTT standard defines fourteen different control packet types. These are CONNECT (1), CONNACK (2), PUBLISH (3), PUBACK (4), PUBREC (5), PUBREL (6), PUBCOMP (7), SUBSCRIBE (8), SUBACK (9), UNSUBSCRIBE (10), UNSUBACK (11), PINGREQ (12), PINGRESP (13) and DISCONNECT (14). [27, p. 16–17] Type in numerical form (used to differentiate packets on protocol level) is presented in parenthesis. Brief listing on what purpose each packet is used for is given below.

- CONNECT, CONNACK and DISCONNECT are used to establish and terminate the actual connection with the broker.
- PUBLISH, PUBACK, PUBREC, PUBREL and PUBCOMP are used when application data is published to broker.
- SUBSCRIBE, SUBACK, UNSUBSCRIBE and UNSUBACK are used when subscriptions are made or canceled.
- PINGREQ and PINGRESP are used to verify that client and broker are alive and reachable.

MQTT packets consist of three parts. These are fixed header, variable header and payload. Fixed header is always present while the existence of the other two depends on a packet. Parts are always organized so that fixed header is first, variable header second and payload third. [27, p. 16] Fixed header contains set of fixed fields while fields in variable header and payload vary between packets.

Text fields in MQTT packets are encoded as UTF-8 (Universal Character Set Transformation Format) strings. Integer values (when two bytes are reserved for the value) are presented using 16 bits and big-endian byte order. [27, p. 13]

Fixed header has three fields. These are packet type, flags and remaining length. Both the packet type and flags reserve four bits. Packet type, as the name suggest, contains packet's type. Flags are specific for each packet (not used in most of the cases). Remaining length specifies how many bytes there are left in the remaining packet (excluding fixed header). It uses variable length encoding and therefore total length of fixed header varies from 2 to 5 bytes. [27, p. 16–20]

Variable header and payload parts vary between packets and because of this they are not discussed here any further. However, parts regarding relevant packets concerning this work are explored in following chapters. Relevant packets concerning this work are: CONNECT, CONNACK, PUBLISH and DISCONNECT. This is minimal set of packets which are required when client needs to establish and terminate connection with broker, and send application data.

3.4.1 CONNECT

A CONNECT packet is sent from client to broker when client establishes connection with broker. Its fixed header part is illustrated in Figure 14. Packet type is 1. Flags are not used. Value and length of Remaining Length field depend on payload part. [27, p. 23]

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type (1)				Reserved			
	0	0	0	1	0	0	0	0
byte 2...	Remaining Length							

Figure 14. Fixed header of *CONNECT* packet [27, p. 23].

Variable header part of *CONNECT* packet is illustrated in Figure 15. It takes ten bytes and contains following fields: Protocol Name, Protocol Level, Connect Flags and Keep Alive. Protocol Name contains “MQTT” string in UTF-8 format. Protocol Level represents protocol’s version number (4 refers to MQTT 3.1.1). [27, p. 23–27] It is followed by number of flags.

	Description	7	6	5	4	3	2	1	0
Protocol Name									
byte 1	Length MSB (0)	0	0	0	0	0	0	0	0
byte 2	Length LSB (4)	0	0	0	0	0	1	0	0
byte 3	'M'	0	1	0	0	1	1	0	1
byte 4	'Q'	0	1	0	1	0	0	0	1
byte 5	'T'	0	1	0	1	0	1	0	0
byte 6	'T'	0	1	0	1	0	1	0	0
Protocol Level									
	Description	7	6	5	4	3	2	1	0
byte 7	Level (4)	0	0	0	0	0	1	0	0
Connect Flags									
byte 8	User Name Flag (1)								
	Password Flag (1)								
	Will Retain (0)								
	Will QoS (01)	1	1	0	0	1	1	1	0
	Will Flag (1)								
	Clean Session (1)								
	Reserved (0)								
Keep Alive									
byte 9	Keep Alive MSB (0)	0	0	0	0	0	0	0	0
byte 10	Keep Alive LSB (10)	0	0	0	0	1	0	1	0

Figure 15. Example of *CONNECT* packet variable header [27, p. 28].

The first two of these flags are related to client’s credentials. If User Name Flag and Password Flag are set to 1, user name and password must be included in payload part. The next three flags are related to client’s last will and concern situation when it disconnects ungracefully. If Will Flag is set to 1, will topic and will message must be included in payload part. Will QoS specifies the QoS level at which broker publishes will message. Will Retain is used to indicate whether broker should retain will message for will topic

when the message is published. Clean Session flag is used to indicate whether client wants to establish clean or restore previous session with broker. Keep Alive, represents time interval in seconds at which broker should receive messages from client. If the client has no application data to be sent, client sends PINGREQ packet. Such mechanism is disabled if Keep Alive value is set to 0. [27, p. 24–28]

The payload part of CONNECT packet contains one or more fields that are prefixed with their length. Client Identifier is only mandatory field. It contains identifier for the client in form of UTF-8 encoded string. [27, p. 29] The following fields exist depending on the flags set in variable header. Will Topic and Will Message are next fields if Will Flag is set to 1. Will Topic contains last will topic in form of UTF-8 encoded string. Will Message contains last will message (in arbitrary format) prefixed with two-byte length. User Name and Password are next fields if User Name Flag and Password Flag are set to 1. User Name contains client's user name in form of UTF-8 encoded string. Password contains client's password (in binary format) prefixed with two-byte length. [27, p. 29–30]

3.4.2 CONNACK

A CONNACK packet is sent from broker to client in response to CONNECT packet. Its fixed header part is illustrated in Figure 16. Packet type is 2. Flags are not used. Total length of CONNACK is four bytes, Remaining Length is therefore set to 2. [27, p. 31]

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet Type (2)				Reserved			
	0	0	1	0	0	0	0	0
byte 2	Remaining Length (2)							
	0	0	0	0	0	0	1	0

Figure 16. Fixed header of CONNACK packet [27, p. 31].

Variable header part of CONNACK packet is illustrated in Figure 17. It takes two bytes and contains following fields: Connect Acknowledge Flags and Connect Return code.

	Description	7	6	5	4	3	2	1	0
Connect Acknowledge Flags		Reserved							SP ¹
byte 1		0	0	0	0	0	0	0	X
Connect Return code									
byte 2		X	X	X	X	X	X	X	X

Figure 17. Variable header of CONNACK packet [27, p. 31–32].

SP (Session Present) flag is set if broker has restored session for client. Other flags are not used. Connect Return code contains result from connection establishment. Standard defines five return codes (0x00, 0x01, 0x02, 0x03, 0x04 and 0x05). In case of accepted

connection Connect Return code is set to 0x00. CONNACK packet does not have payload part. [27, p. 32–33]

3.4.3 PUBLISH

A PUBLISH packet is sent when the client sends application data to broker or when broker sends application data to client. Its fixed header part is illustrated in Figure 18. Packet type is 3. DUP flag is set if packet is re-delivered (sent again). QoS level, as the name suggest, specifies QoS level for the packet. RETAIN is set if client requests broker to store application data as last good value for topic. Value and length of Remaining Length field depend on payload part which in this case contains application data. [27, p. 33–34]

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type (3)				DUP flag	QoS level		RETAIN
	0	0	1	1	X	X	X	X
byte 2	Remaining Length							

Figure 18. Fixed header of PUBLISH packet [27, p. 33].

Variable header part of PUBLISH packet is illustrated in Figure 19. It takes seven bytes and contains following fields: Topic Name and Packet Identifier.

	Description	7	6	5	4	3	2	1	0
Topic Name									
byte 1	Length MSB (0)	0	0	0	0	0	0	0	0
byte 2	Length LSB (3)	0	0	0	0	0	0	1	1
byte 3	'a' (0x61)	0	1	1	0	0	0	0	1
byte 4	'/' (0x2F)	0	0	1	0	1	1	1	1
byte 5	'b' (0x62)	0	1	1	0	0	0	1	0
Packet Identifier									
byte 6	Packet Identifier MSB (0)	0	0	0	0	0	0	0	0
byte 7	Packet Identifier LSB (10)	0	0	0	0	1	0	1	0

Figure 19. Example of PUBLISH packet variable header [27, p. 35–36].

Topic Name contains topic in form of UTF-8 encoded string. Packet Identifier contains identifier (integer) for packet. Payload part contains application data which can be in arbitrary format. [27, p. 35–36]

3.4.4 DISCONNECT

A DISCONNECT packet is sent from client to broker when client terminates connection with it. Its fixed header part is illustrated in Figure 20. Packet type is 14. Flags are not used. Total length of DISCONNECT is two bytes, Remaining Length is therefore set to 0. [27, p. 49]

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type (14)				Reserved			
	1	1	1	0	0	0	0	0
byte 2	Remaining Length (0)							
	0	0	0	0	0	0	0	0

Figure 20. Fixed header of *DISCONNECT* packet [27, p. 49].

DISCONNECT packet does not have variable header or payload parts. In contrast to CONNECT packet, DISCONNECT packet has no response counterpart.

4. CODESYS

This chapter presents the core concepts behind CODESYS focusing particularly on its programming aspects. API which CODESYS provides for socket programming is especially studied. It has central role in this work for implementing upper layer MQTT protocol presented in previous chapter.

4.1 Overview

CODESYS (COntroller DEvelopment SYStem) is commercial software product developed by Smart Software Solutions GmbH. It is aimed for developing manufacturer independent IEC (International Electrotechnical Commission) 61131-3 applications for CODESYS compatible controllers and SoftPLCs. [43] Current version of CODESYS is prefixed with V3.5 while older version starts with V2.3. Older version is maintained with bug fixing until the end of 2019. [44]

According to CODESYS device directory which lists CODESYS compatible devices, there are currently over 300 supported devices from different manufacturers. These range from various controllers to PC based display panels with touch screen. [45]

4.2 Components

CODESYS is manifold software product. It can be perceived as consisting of engineering and device levels. These are illustrated in Figure 21.

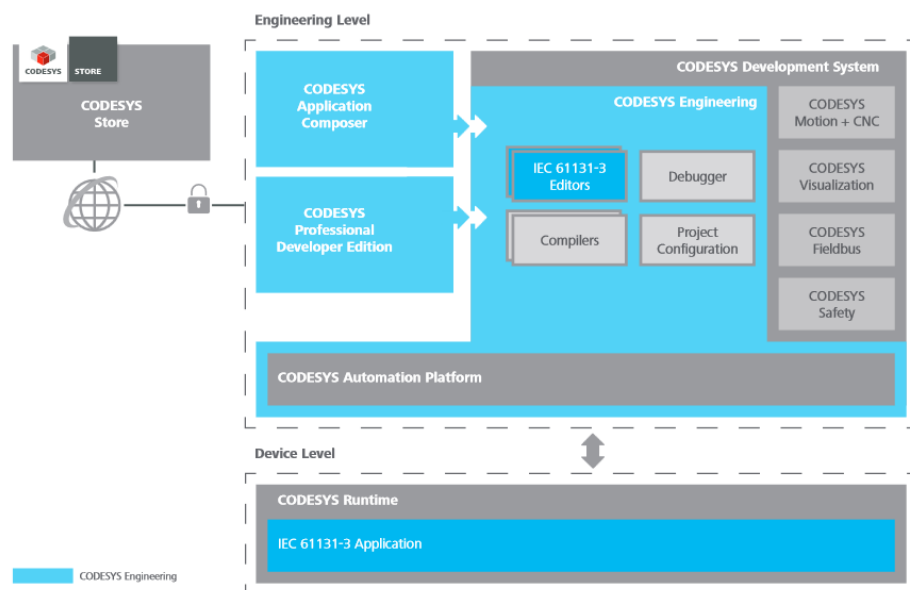


Figure 21. Overview of CODESYS components [46, p. 2].

CODESYS Development System is the core of engineering part. It is essentially an IDE (Integrated Development Environment) which is used to develop manufacturer independent IEC 61131-3 applications that run on CODESYS compatible devices. Development System contains code editors, compilers, debugger and various other components which handle different aspect of software development, project management and target device hardware configuration. It also allows to develop HMIs (Human Machine Interface), configure and program various fieldbus interfaces, and build safety and motion control applications under single tool. [46, p. 4–6] CODESYS Development System is available for Windows and comes equipped with SoftPLC (CODESYS Control Win) as trial target system for direct testing of developed CODESYS applications [47].

By default, Development System offers set of ready-made libraries (functions, function blocks, etc.) for various purposes such as TCP/IP communication. More libraries can be downloaded from CODESYS Store which is market place for CODESYS related software, libraries and add-ons [46, p. 7].

Another major part of CODESYS is CODESYS Runtime. It is software system that runs on target device (e.g. in PLC) making it compatible with CODESYS. Runtime is responsible for executing actual application code and communicating with Development System. Communication includes receiving compiled application code in binary format and debugging its execution. Handling I/O systems and field busses (updating process image) is also responsibility of Runtime. [48, p. 4]

Runtime is available for various CPU (Central Processing Unit) architectures (e.g. Intel 80x86, ARM (Advanced Reduced Instruction Set Computer Machines) and Power) and platforms (e.g. Windows and Linux). Its functionality is distributed into various components that makes its structure modular. Adaption of the Runtime to target platform is done with CODESYS Control Runtime Toolkit software. It allows to select and configure existing components, software modules and drivers for desired functionality and compile them to executable Runtime environment. Modularity means that device manufacturer can select only desired components and does not need to implement support for all of them in order to make target device CODESYS compatible. [48, p. 4–7]

4.3 Programming

PLC programming in CODESYS is done using languages defined in IEC 61131-3 standard. The standard defines five distinct programming languages which are illustrated next chapter in more detail. In addition, API for socket programming in CODESYS is also presented. It is vital in terms of this work since it is used as a foundation for implementing upper-layer MQTT protocol.

4.3.1 IEC 61131-3 standard and programming languages

IEC 61131-3 standard defines five individual languages for PLC programming. It is third part of IEC 61131 which is an international standard for industrial programmable controllers. On top of languages, IEC 61131-3 addresses also PLC program execution, addressing, data formats/data structures, use of symbols, sequential control and connections between languages [49, p. 136]. First version of the standard was published in 1993. It was followed by two latter versions: the first was published in 2002 and second (which is also the latest version) in 2013. [49, p. 134] The standard is not public but is available for purchase [50].

One of the main goals of IEC 61131-3 is to establish common ground for PLC programming and ease program development between various manufacturer specific environments. However, although the standard unifies languages, it does not specify set of absolute rules which manufacturers must precisely follow; it is more like list of comprehensive guidelines against which manufacturers point their compliance with the standard. As such there can be relatively large differences between different manufacturers regarding programming. E.g. code written in one environment might not look the same as code written in another environment although the language is same. [49, p. 136–137]

IEC 61131-3 defines two textual languages and three graphical languages. Textual languages are [49, p. 134]:

- Instruction List (IL) and
- Structured Text (ST).

Graphical languages are [49, p. 134]:

- Ladder Diagram (LD),
- Function Block Diagram (FBD) and
- Sequential Function Chart (SFC).

Programs, written with these languages, are organized using POU (Program Organization Unit). IEC 61131-3 defines four different type of POU which are Program, Function Block (FB), Function (F) and Class. POU are treated as independent, reusable program units which implement certain functionality of the program. These units can be nested relative to each other and called hierarchically according to program logic. [49, p. 143–145] POU which constitute PLC programs can be implemented with any of the languages defined in the standard. These languages are next briefly illustrated as they appear in CODESYS V3.5 IDE.

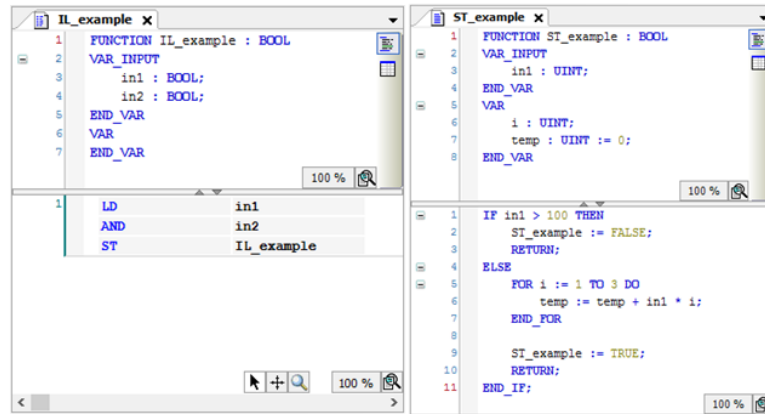


Figure 22. Illustration of Instruction List (left) and Structured Text (right) languages in CODESYS.

IL is illustrated left in Figure 22. It is assembler-like textual language which contains set of primitive instructions that are used to implement desired functionality. IL is considered low level language compared to languages such as ST or C. ST is illustrated right in Figure 22. It contains features like loops, conditionals and high data abstraction which are considered typical for high level languages. Syntax of ST resembles Pascal.

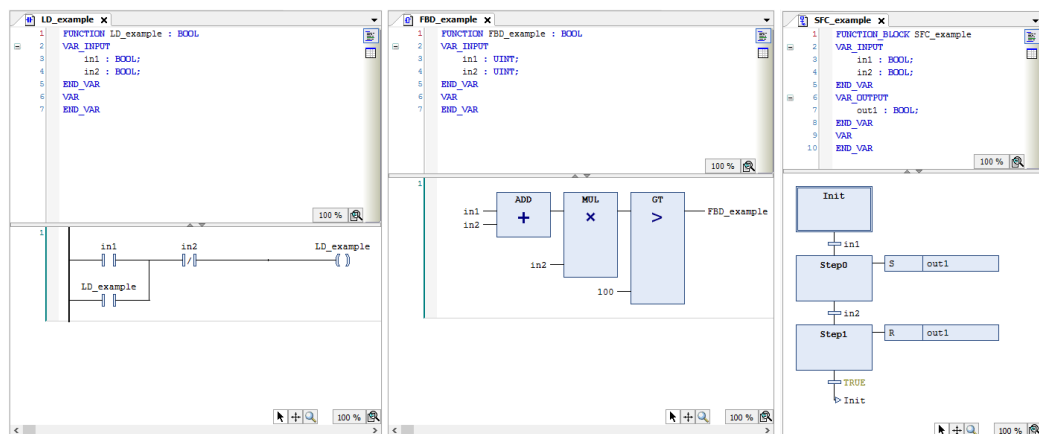


Figure 23. Illustration of Ladder Diagram (left), Function Block Diagram (middle) and Sequential Function Chart (right) languages in CODESYS.

LD is illustrated left in Figure 23. It resembles relay wiring diagram in which variables are presented as relay contacts and coils. Program logic in LD is constructed by forming connections between contacts and coils. FBD is illustrated middle in Figure 23. It in turn resembles wiring diagram found in digital electronics. Program logic in FBD is constructed by connecting various function blocks together which take care of certain sub function in the program. Third graphical language is SFC. It is illustrated right in Figure 23. Essential components of SFC are steps and transitions. Steps are associated with actions (e.g. setting outputs) which take place when the step is active. The program logic in SFC is constructed by connecting these steps together and by defining conditions for transition between them.

4.3.2 Socket programming API

Sockets are network endpoints in computer programs which facilitate communication between networked applications. They provide framework for programs to send and receive data over network connection. This chapter presents CODESYS socket programming API. It is used as a foundation for implementing MQTT protocol on top of TCP/IP stack in this work. CODESYS V3.5 provides multiple socket APIs which are available through various system libraries (SysSocket23, SysSocket and SysSocketAsync). This work utilizes SysSocket23 library. It provides necessary API for implementing TCP client functionality and is illustrated next with ST. CODESYS has no API for TLS.

Before any application data can be transmitted between client (MQTT client) and server (MQTT broker), client must create socket and establish connection to server using created socket. With SysSocket23 library, socket is created using *SysSockCreate(diAddressFamily, diType, diProtocol)* function. It takes following parameters [51, p. 3–5].

- *diAddressFamily* (DINT). Specifies what type of address scheme and what type of protocols are used with socket. In case of Internet communication, parameter is set to `SOCKET_AF_INET` (equivalent to 2) which refers to IPv4 (Internet Protocol version 4) and which allows transfer data with TCP.
- *diType* (DINT). Specifies what form of communication is associated to socket. In case of connection oriented communication (TCP) parameter is set to `SOCK_STREAM` (equivalent to 1).
- *diProtocol* (DINT). Specifies what protocol is used between client and server. In case of TCP it is set to `SOCKET_IPPROTO_TCP` (equivalent to 6).

SysSockCreate(...) returns handle to socket which is used in subsequent function calls related to the socket. If socket creation fails `SOCKET_INVALID` (equivalent to -1) is returned. Following code illustrates usage of *SysSockCreate(...)* when it is used to create TCP socket.

```
VAR
    diSocket : DINT := SOCKET_INVALID;
    diAddressFamily : DINT := SOCKET_AF_INET;
    diType : DINT := SOCKET_DGRAM;
    diProtocol : DINT := SOCKET_IPPROTO_TCP;
END_VAR

//Create TCP socket
diSocket := SysSockCreate(diAddressFamily, diType, diProtocol);
```

After socket is created, client can connect to server using *SysSockConnect(diSocket, pSockAddr, diSockAddrSize)* function. It takes following parameters.

- *diSocket* (DINT). Specifies socket handle which was returned by *SysSockCreate(...)* function.

- *pSockAddr* (DWORD). Specifies address of SOCKADDRESS data structure which contains server's IP address and TCP port.
- *diSockAddrSize* (DINT). Specifies size of SOCKADDRESS data structure. SOCKADDRESS must be prepared accordingly before it is given to *SysSockConnect(...)*.

On success *SysSockConnect(...)* returns 1, otherwise 0. Following code illustrates process of preparing SOCKADDRESS data structure and using *SysSockConnect(...)* to connect to TCP server.

```

VAR
    diSocket : DINT := SOCKET_INVALID;
    sockAddr : SOCKADDRESS;
    sockAddr.sin_family := SOCKET_AF_INET;
    sockAddr.sin_addr := SysSockInetAddr('192.168.0.1');
    sockAddr.sin_port := SysSockHtons(1883);
    result : BOOL;
END_VAR

//Create TCP socket
//...

//Connect to server
result := SysSockConnect(diSocket, ADR(sockAddr), SIZEOF(sockAddr));

```

When client has successfully established connection with server it can send and receive data (MQTT control packets) through the socket. Sending of data is done using *SysSockSend(diSocket, pbyBuffer, diBufferSize, diFlags)* function and receiving using *SysSockRecv(diSocket, pbyBuffer, diBufferSize, diFlags)* function. Both functions take following parameters [52, p. 11–13].

- *diSocket* (DINT). Specifies socket handle which was returned by *SysSockCreate(...)* function.
- *pbyBuffer* (DWORD). Specifies address of byte array which is used for storing bytes that are either received or sent through the socket.
- *diBufferSize* (DINT). Specifies size of the byte array.
- *diFlags* (DINT). Specifies way the function should be called (usually set to 0).

By default all the socket function calls are blocking (e.g. *SysSockRecv(...)* blocks program execution until new data is received) which might not be desired property when PLC program requires some minimum time to update its process image. This can be solved by setting socket to non-blocking mode with *SysSockIoctl(...)* which results in socket function calls except *SysSockConnect(...)* to be non-blocking. *SysSockConnect(...)* is always blocking. Other solutions to avoid blocking behavior would be to use *SysSockSelect(...)* or use separate tasks within PLC program. [30, p. 11] [51, p. 10] These methods are not covered here. *SysSockIoctl(diSocket, diCommand, piParameter)* takes following parameters [52, p. 11].

- *diSocket* (DINT). Specifies socket handle which was returned by *SysSockCreate(...)* function.
- *diCommand* (DINT). Specifies the command to apply to the socket.
- *piParameter* (DWORD). Specifies address of command parameter.

SysSockSend(...) returns a number of sent bytes. In case of error, -1 is returned. When connection is closed by peer, 0 is returned. *SysSockRecv(...)* returns number of received bytes. When data is not available and socket is set to non-blocking, -1 is returned. When connection is closed by peer, 0 is returned. Following code illustrates usage of *SysSockIoctl(...)*, *SysSockSend(...)* and *SysSockRecv(...)*.

```

VAR
    diSocket : DINT := SOCKET_INVALID;
    buffer : ARRAY [0..127] OF BYTE;
    IoctlParameter : DINT := 1;
    result : DINT;
VAR_END

//Create TCP socket and connect to TCP server
//...

//Set socket to non-blocking mode
SysSockIoctl(diSocket, SOCKET_FIONBIO, ADR(IoctlParameter));
//Send data to server
result := SysSockSend(diSocket, ADR(buffer), SIZEOF(buffer), 0);
//Receive data from server
result := SysSockRecv(diSocket, ADR(buffer), SIZEOF(buffer), 0);
//Close connection
SysSockClose(diSocket);

```

Connection with server is closed with *SysSockClose(diSocket)*. It takes socket handle as its only parameter.

5. IMPLEMENTATION

This chapter presents the details in terms of actual implementation done in the work. In addition to CODESYS MQTT client, overview of the whole system and its components are also provided. Each component is covered in its own sub chapter. In addition, various configurations to which these components can be organized in are also discussed.

5.1 Overview of the system

Main goal of developed CODESYS MQTT client is to make it possible to transfer data from CODESYS compatible device to IoT-Ticket. IoT-Ticket has support for MQTT protocol via separate adapter (MQTT-REST adapter) which acts as an intermediate node between MQTT client and IoT-Ticket's REST interface. As the name suggests, adapter converts MQTT messages to HTTPS (Hypertext Transfer Protocol Secure) requests which are compatible with IoT-Ticket's REST interface. MQTT-REST adapter is client for MQTT broker (through which MQTT messages are received) and client for IoT-Ticket (to which received data is finally sent). Figure 24 illustrates the relationship of CODESYS MQTT client, MQTT broker, MQTT-REST adapter and IoT-Ticket.

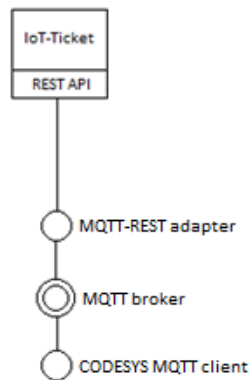


Figure 24. *System overview.*

When data is sent from CODESYS MQTT client to IoT-Ticket, CODESYS MQTT client prepares PUBLISH packet and sends it to MQTT broker (payload portion contains the actual data). The packet is sent with predefined topic which is also known and subscribed by MQTT-REST adapter. This allows MQTT broker to deliver received packet (or more precisely its payload) from CODESYS MQTT client to MQTT-REST adapter. When MQTT-REST adapter receives the packet, it extracts its payload, generates HTTPS requests based on the contents and sends these requests to IoT-Ticket's REST interface. These are essentially the steps which are involved when CODESYS MQTT client sends data to IoT-Ticket.

5.2 CODESYS MQTT client

This chapter presents the outcome of the work in terms of developed CODESYS MQTT client regarding its features, usage, architecture and improvements. Client was developed as an external CODESYS library using ST language with version V3.5 SP8 Patch 4 of CODESYS IDE. It is compatible with MQTT 3.1.1 and is built on top of non-blocking TCP sockets.

5.2.1 Features

Developed CODESYS MQTT client supports only clean sessions when establishing connection with broker. This means that broker will not store any state for the client. Authentication with user name and password, keep alive mechanism and last will feature are also not supported. When publishing data to topic, PUBLISH packets are always sent with QoS level 0 (QoS 1 and 2 are not supported by the implementation) and with retain flag set to false (broker will not retain published data for topic as last good value). Developed client can only publish data to topics, subscribing topics is not supported. Main features of developed client are listed below.

- MQTT 3.1.1 compatible.
- Utilizes non-blocking TCP sockets.
- Can publish messages only with QoS 0.
- No support for subscribing topics.
- No support for authentication.
- No support for keep alive.
- Supports payload encryption using RC4 and integrity check using SHA-256.

As a security feature, developed CODESYS MQTT client provides means to encrypt PUBLISH packets' payload and equip it with mechanism to check payload's integrity at receiving end. When enabled, client calculates one-way hash using SHA-256 algorithm from the payload and appends the result to the end of payload. Payload and hash are then encrypted together using RC4 algorithm and shared secret key. After encryption data is then sent to MQTT-REST adapter via MQTT broker.

At receiving end payload is decrypted and accompanied hash extracted from it. Extracted hash is checked against hash which is independently calculated by receiver from received payload. If they match, received payload is considered intact. By this manner both, integrity and confidentiality (encryption) are obtained. Also authenticity of the sender is guaranteed because only sender which has access to the secret key is able to produce payload which is considered valid at the receiver. In other words, it is very unlikely that during the transmission someone has been able to alter contents of the message so that accompanying hash would match at receiving end. Virtually this would mean that secret key which is used to encrypt the payload is known by intermediate attacker. Therefore, if

payload is captured, then modified and sent to MQTT-REST adapter, it will not pass integrity check. Process of encryption, decryption and integrity check are illustrated in Figure 25.

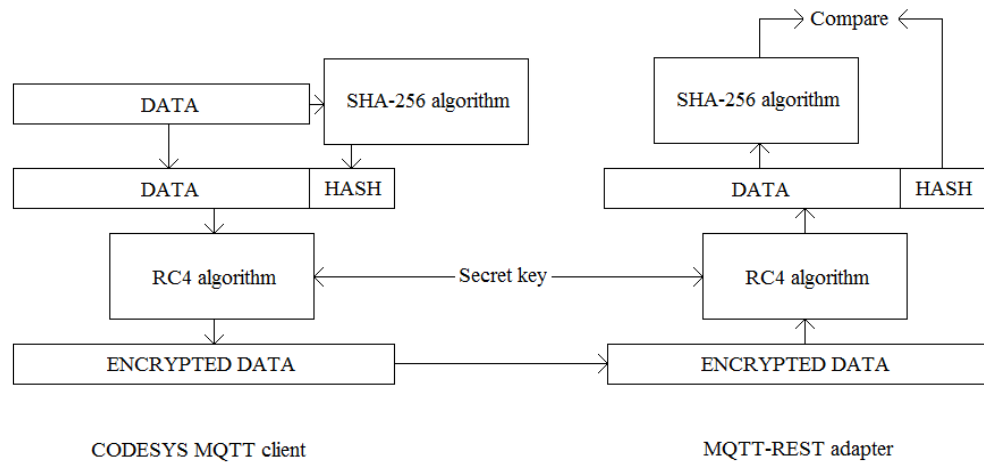


Figure 25. Encryption, decryption and integrity check.

With payload encryption and integrity check illustrated in Figure 25 there is no protection against replay attacks. This means that if payload is captured during transmission and sent at any given time to MQTT-REST adapter as it is, it will pass the integrity check and therefore get treated as a valid payload.

5.2.2 Usage

MQTT client library can be installed to CODESYS IDE through *Library repository*. After installation, library is available in *Library Manager* from where it can be added to CODESYS project. It contains multiple POU's from which *MQTTClient* function block is the actual MQTT client. The library contains also set of helper functions, constants and one enum which lists MQTT client's error codes. Library requires SysSocket23 (3.5.8.0) and SysStr23 (3.5.8.0) libraries as dependencies.

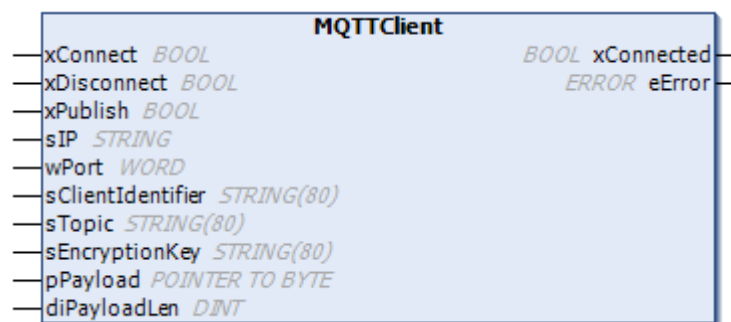


Figure 26. CODESYS MQTT client function block.

Figure 26 illustrates appearance of MQTT client function block in CODESYS IDE. It has ten inputs and two outputs. They are listed and explained in more detail in Table 1.

Table 1. CODESYS MQTT client's function block interface.

Scope	Name	Type	Initial	Comment
Input	xConnect	BOOL		Connect to broker
	xDisconnect	BOOL		Disconnect from broker
	xPublish	BOOL		Publish to topic
	sIP	STRING		IP address of the broker
	wPort	WORD		TCP port of the broker
	sClientIdentifier	STRING		Identifier for MQTT client (max. 80 characters)
	sTopic	STRING		Topic for MQTT publish message (max. 80 characters)
	sEncryptionKey	STRING		Encryption key for RC4 payload encryption (max. 80 characters)
	pPayload	POINTER TO BYTE		Payload for MQTT publish message (max. 891 bytes if encryption is not used, max. 859 bytes if used)
	diPayloadLen	DINT		Payload length
Output	xConnected	BOOL	FALSE	True when successfully connected to MQTT broker
	eError	ERROR	0	Error code

Before a connection can be established with MQTT broker using the function block, broker's IP address and TCP port must be provided via *sIP* and *wPort* inputs. An identifier for the client must also be given beforehand. It is provided via *sClientIdentifier* input.

Connection is established with MQTT broker by providing rising edge to *xConnect* input. The program execution blocks during connection establishment. In order to disconnect from the MQTT broker, rising edge to *xDisconnect* input must be provided. To send PUBLISH packet to MQTT broker, rising edge to *xPublish* input must be provided.

Topic to which PUBLISH packets are published, is provided via *sTopic* input. Payload for PUBLISH packet is provided via *pPayload* input. *pPayload* is supplied as an address to byte array. Length of the byte array is specified via *diPayloadLen* input.

RC4 payload encryption and integrity check are enabled when encryption key is provided via *sEncryptionKey*. Encryption key is used to encrypt the payload with RC4 algorithm. When encryption key is provided, maximum size of the payload is decreased from 891

bytes to 859 bytes. This is because calculated SHA-256 hash takes 32 additional bytes from payload portion.

Table 2. List of CODESYS MQTT client's error codes.

Name	Value	Comment
NO_ERROR	0	No error
INVALID_CLIENT_IDENTIFIER	100	Invalid client identifier input parameter (does not exist or is too big)
INVALID_TOPIC	101	Invalid topic input parameter (does not exist or is too big)
INVALID_PAYLOAD	102	Invalid payload input parameter (is too big)
SOCKET_INVALID	200	TCP socket error
UNABLE_TO_CONNECT	201	Client is unable to connect to MQTT broker
UNABLE_TO_SEND_DATA	202	Client is unable to send data through TCP socket
UNABLE_TO_RECEIVE_DATA	203	Client is unable to receive data from TCP socket
TX_OVERFLOW	300	Client's transmit buffer has overflowed
RX_OVERFLOW	301	Client's receive buffer has overflowed
PAYLOAD_OVERFLOW	302	Client's payload buffer has overflowed
MQTT_CONNECTION_REFUSED	400	Broker has refused client to connect
MQTT_CONNECTION_REFUSED_0x01	401	Broker has refused client to connect (MQTT error code 0x01)
MQTT_CONNECTION_REFUSED_0x02	402	Broker has refused client to connect (MQTT error code 0x02)
MQTT_CONNECTION_REFUSED_0x03	403	Broker has refused client to connect (MQTT error code 0x03)
MQTT_CONNECTION_REFUSED_0x04	404	Broker has refused client to connect (MQTT error code 0x04)
MQTT_CONNECTION_REFUSED_0x05	405	Broker has refused client to connect (MQTT error code 0x05)
INVALID_MQTT_PACKET	406	Received invalid MQTT packet

When CODESYS MQTT client has successfully connected to MQTT broker, *xConnected* output is set as true (false whenever disconnected). If error occurs in any state of operation, error code is provided via *eError* output. Table 2 lists all error codes.

5.2.3 Architecture

When CODESYS MQTT client library is added into CODESYS project, *MQTTClient* function block becomes available. It implements actual MQTT client functionality and consists of set of methods. Each method implements part of the overall functionality utilizing helper functions which are provided by the library. Library defines also set of constants which specify boundaries for supported MQTT packet sizes and one enum (type of *ERROR*) which lists *MQTTClient*'s error codes. Figure 27 lists all POUs implemented in this work for CODESYS MQTT client library.

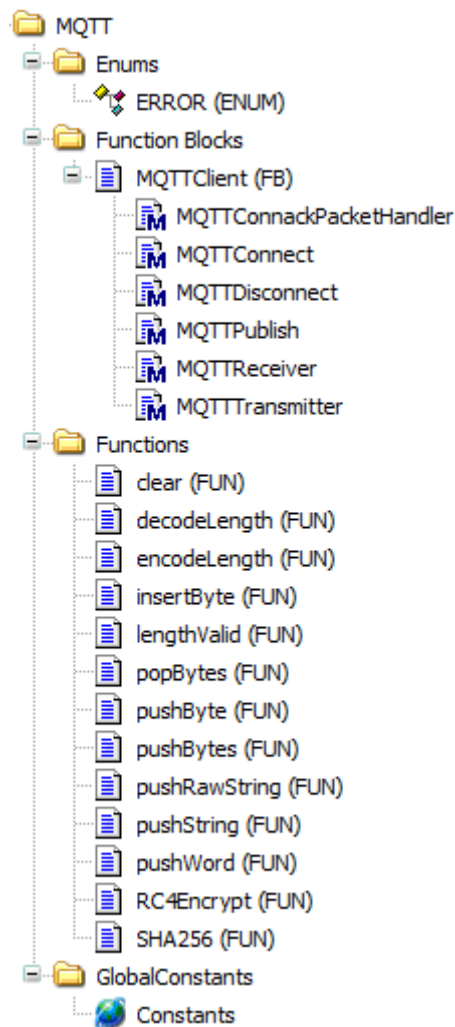


Figure 27. CODESYS MQTT client library's POUs.

MQTTClient function block has, in total, six methods and is responsible for calling them accordingly. *MQTTConnect* method is called by function block when rising edge to *xConnect* input is provided. *MQTTConnect* creates TCP socket and uses it to establish connection to MQTT broker. It also creates CONNECT packet and stores it to *MQTTClient*'s transmit buffer. All packets (except DISCONNECT packet) are first stored in *MQTTClient*'s transmit buffer from where they are sent to MQTT broker. *MQTTTransmitter* method is responsible for sending packets stored in transmit buffer to MQTT broker.

MQTTReceiver method is in turn responsible for receiving data from MQTT broker and storing it in *MQTTClient*'s receive buffer. It also parses packets from receive buffer and passes them to right handler when they are complete. *MQTTConnackPacketHandler* method is the only implemented handler and is responsible for handling CONNACK packets and determining if the connection is accepted by MQTT broker or not. *MQTTTransmitter* and *MQTTReceiver* are called as long as the client is connected to the broker and as long as *MQTTClient* is itself called. *MQTTPublish* method is called by *MQTTClient* when rising edge to *xPublish* input is provided. It takes care of creating PUBLISH packet and storing it to *MQTTClient*'s transmit buffer from where it is sent to MQTT broker. If payload encryption is enabled (encryption key is provided via *sEncryptionKey*) creation of PUBLISH packet differs in such way that *MQTTPublish* calculates SHA-256 hash from provided payload using *SHA256* helper function and appends it to the end of provided payload before it encrypts them together with *RC4Encrypt* helper function. Otherwise creation and storage of PUBLISH packet is the same. *MQTTDisconnect* method is called by *MQTTClient* when rising edge to *xDisconnect* input is provided. It creates DISCONNECT packet and sends it immediately to broker and closes the socket that was opened by *MQTTConnect*. DISCONNECT packets are not stored in *MQTTClient*'s transmit buffer.

5.2.4 Improvements

Developed CODESYS MQTT client is capable of publishing application data to desired topic. In most cases this is sufficient when data is only written to IoT-Ticket. If data is also required to be read from IoT-Ticket, there is a problem since CODESYS MQTT client has no support for receiving application data. This could be solved by implementing subscribe functionality to CODESYS MQTT client. However, MQTT-REST adapter would also require some changes since it currently treats every received payload as write operation. (Payload has specific format and parameters which currently constitute write operation.) Therefore, new payload format and set of new rules would be required since read operation is somewhat different from write operation. Some extra logic for the response handling would be also required since response from MQTT-REST adapter must be delivered to CODESYS MQTT client. To accomplish this, the client would need to subscribe specific topic to which the response is sent from MQTT-REST adapter. In addition, if encryption and integrity check are required also when receiving response, decryption and integrity check functionalities would be needed also on CODESYS MQTT client side. They are currently implemented only on MQTT-REST adapter side.

Having support just for subscribing topics would be major improvement in overall sense. Some other possible improvements which could be implemented to CODESYS MQTT client are indicated below.

- Support for publishing data with QoS level 1 and 2.
- Support for setting retained flag when publishing data to topic.

- More versatile options regarding connection establishment: support for last will feature, persistent session, credentials (user name and password) and keep alive feature.
- Enhancements to payload encryption used between CODESYS MQTT client and MQTT-REST adapter.

These improvements would require substantial changes to be made internally in CODESYS MQTT but also to its external interface. Currently the implementation has ten individual inputs. With mentioned improvements the number could go easily over fifteen. When the number of inputs increases, the interface gets more complicated and FB itself becomes more complex to utilize in the program code. This is especially the case with graphical languages defined in IEC 61131-3. To reduce this effect CODESYS MQTT client could be divided into smaller FBs which each would implement part of the overall functionality. For example, connection establishment could be done in one these FBs and be equipped only with inputs which are relevant in terms of connection establishment. Carrying the idea further, publish and subscribe functionalities could be also divided into separate FBs and be equipped only with inputs which are relevant in terms of their functionality. This would mean lower input count per FB and in this regard reduce the complexity of CODESYS MQTT client interface and make it easier to utilize.

Payload encryption used between CODESYS MQTT client and MQTT-REST adapter would also require some improvements. First of all, it is based on symmetric RC4 cipher which is considered somewhat obsolete and should be therefore replaced. RC4 was chosen as a cipher due to its simplicity and because it is fast. A good option for replacing RC4 would be popular AES cipher.

Another security related issue is that there is not protection against replay attacks when payload encryption is used (also the case when not used). This is also a clear issue since MQTT-REST adapter is not able to distinguish valid payload from replayed one. Such replay attack could be even realized with relative ease. It just requires establishing connection with MQTT broker and subscribing same topic with MQTT-REST adapter. If such conditions are met, replay attack could be performed simply by publishing data back to topic from where it was received. This has the effect that MQTT-REST adapter receives replayed data and is not aware of that.

To solve this, access to MQTT broker could be restricted only to well-known clients. In practice this would mean use of credentials within CONNECT packet when connecting to MQTT broker. However, what makes this approach unfavorable is the fact that CODESYS MQTT client has support only for plaintext TCP. This means that credentials would be sent in plaintext and would be visible for any intermediate observer implying that restricting clients just with credentials is not sustainable solution.

Another effort to prevent replay attacks could be to include additional information into PUBLISH packet's payload portion which would allow distinguishing packets from each

other. In practice this would mean some sort of sequence numbers that would be incremented in every sent PUBLISH packet. By this manner if payload gets replayed, MQTT-REST adapter would recognize it from the sequence number. However, utilizing sequence numbers would require extra logic in both CODESYS MQTT client and MQTT-REST adapter.

5.3 MQTT broker

MQTT broker is responsible for delivering application data (payload) in PUBLISH packets from CODESYS MQTT client to MQTT-REST adapter and managing client connections. MQTT broker which is used in this work is an open source Mosquitto broker. It supports both, TCP and TLS.

When TLS is used, Mosquitto must be equipped with suitable certificates and private key. Locations to these certificates and private key are specified in its configuration file, which by default in most Linux distributions is located in */etc/mosquitto/mosquitto.conf*. Example configuration when both TCP and TLS are used, is illustrated below.

```
pid_file /var/run/mosquitto.pid
persistence true
persistence_location /var/lib/mosquitto/
log_dest file /var/log/mosquitto/mosquitto.log
include_dir /etc/mosquitto/conf.d

listener 1883

listener 8883
cafile /etc/mosquitto/ca_certificates/ca.crt
certfile /etc/mosquitto/certs/server.crt
keyfile /etc/mosquitto/certs/server.key
require_certificate true
```

listener 1883 specifies connection point for TCP connections and *listener 8883* connection point for TLS connections. TLS listener requires two certificates and one private key. Paths to these certificates and private key are shown in the above configuration. When *require_certificate* is set to *true*, Mosquitto broker requires certificate from client (in TLS vocabulary this is called mutual authentication).

CA (Certificate Authority) certificate (*ca.crt*) contains CA's public key. It is used to verify validity of client certificate which is sent from client if mutual authentication is requested. Broker's own certificate (*server.crt*) is in turn provided to client whenever client is connecting to broker. *server.key* is broker's private key and is used to decrypt messages during TLS handshake procedure. These messages are sent from client to broker and are encrypted with corresponding public key (client gets this public key from *server.crt*).

When TLS is used client is also provided with two certificates and one private key. One of these certificates is the same CA certificate (*ca.crt*). At client side it is used to verify

validity of received broker certificate (*server.crt*). Another certificate is client's own certificate. It is sent to broker if mutual authentication is requested. Provided private key is client's own private key which is used to decrypt messages that are encrypted with corresponding public key.

5.4 MQTT-REST adapter

MQTT-REST adapter is based on existing implementation which is developed at Wapice. It was modified in terms of this work so that support for following features were obtained.

- Payload encryption and integrity check support for payload received from CODESYS MQTT client.
- TLS support for data transfer between MQTT broker and MQTT-REST adapter.

Main goal of MQTT-REST adapter is to convert payload in PUBLISH packet to HTTPS requests which are valid against IoT-Ticket's REST interface. To accomplish this, MQTT-REST adapter needs to know under which enterprise, device and data node received data is stored in IoT-Ticket. Without this information, adapter cannot convert received PUBLISH packets to valid HTTPS requests.

Part of the required information is embedded into adapter's configuration file which contains user name and password that specify the enterprise under which the data is stored in IoT-Ticket. Rest of the required information is embedded into PUBLISH packet's payload portion together with the actual data. Payload which is accepted by MQTT-REST adapter is formatted using CSV and contains following fields.

```
deviceName;deviceManufacturer;dataNodeName;dataNodePath;value;timestamp;unit;datatype
```

Whenever CODESYS MQTT client wishes to send data to MQTT-REST adapter, it must use format illustrated above. Otherwise MQTT-REST adapter is not able to generate valid data node write request.

First two fields concern the device under which the data node is stored in IoT-Ticket. Remaining fields concern the data node itself. Data node holds the actual data (*value*) in addition to other associated information (*dataNodeName*, *dataNodePath*, *timestamp*, *unit* and *datatype*). *deviceName*, *deviceManufacturer*, *dataNodeName* and *value* are mandatory fields. Other fields are optional.

When adapter receives PUBLISH packet, it checks that exactly eight fields are present and mandatory fields are not empty. If these conditions are met, adapter reads values from these fields and generates required HTTPS requests based on their values. Adapter takes care of registering devices in case they do not exist on IoT-Ticket side.

If RC4 payload encryption is used at CODESYS MQTT client side, adapter must be informed about this. Whether the encryption is used or not is specified via flag in adapter's

configuration file. When the flag is set to 1 adapter assumes that every received payload is encrypted with RC4 using shared secret key and contains additional hash in the end of payload calculated using SHA-256 algorithm. Secret key is also provided via this configuration file.

If the flag is set to 1, adapter first decrypts received payload using provided secret key and extracts hash (32 last bytes of the payload) from it. Adapter then calculates corresponding hash by itself and compares it with the received one. If they match payload is considered valid and it is processed in the same way as noted earlier. If there is mismatch between received and calculated hash, payload is ignored and not processed further.

Whether the TLS is used between adapter and MQTT broker is specified with another flag. When set to 1 adapter will use TLS as an underlying protocol in all communication with MQTT broker. Implementation utilizes mutual authentication so both endpoints must be equipped with suitable certificates before working TLS connection can be established. Example configuration is illustrated below.

```
iot.url=https://my.iot-ticket.com/api/v1
iot.username=iot_user
iot.password=iot_password

broker.tls.enable=0
broker.url=tcp://127.0.0.1:1883
broker.topic.sub=subscribe_topic
broker.credentials.enable=0
broker.credentials.username=
broker.credentials.password=

payload.decryption.enable=1
payload.decryption.key=decryption_key
```

iot.url is the address of IoT-Ticket's REST API (*https://my.iot-ticket.com/api/v1*). *iot.username* and *iot.password* are username and password for IoT-Ticket. *broker.tls.enable* is set to 0 when TCP is used between MQTT-REST adapter and MQTT broker. When *broker.tls.enable* is set to 0, *broker.url*, which specifies MQTT broker's IP address and TCP port, is required to be set to *tcp://{ip address}:1883* where *{ip address}* is the IP address of MQTT broker. TCP port is set to 1883 which MQTT broker uses for TCP connections. *broker.tls.enable* is set to 1 when TLS is used between MQTT-REST adapter and MQTT broker. When *broker.tls.enable* is set to 1, *broker.url* is required to be set to *ssl://{ip address}:8883* where *{ip address}* is the IP address of MQTT broker. TCP port is set to 8883 which MQTT broker uses for TLS connections. Use of credentials when connecting to MQTT broker is enabled when *broker.credentials.enable* is set to 1. Credentials are supplied via *broker.credentials.username* and *broker.credentials.password*.

5.5 System configurations

System components (CODESYS MQTT client, MQTT broker and MQTT-REST adapter) can be organized into different configurations depending on the use case. Three possible configurations are illustrated in Figure 28. Each has slightly different approach to deliver data over unsecure communication channel from CODESYS MQTT client to IoT-Ticket. Configurations consist of top and bottom parts. Top part is considered as IoT-Ticket side and bottom part as enterprise side. Parts communicate together over the Internet. Goal is transfer data from enterprise side to IoT-Ticket.

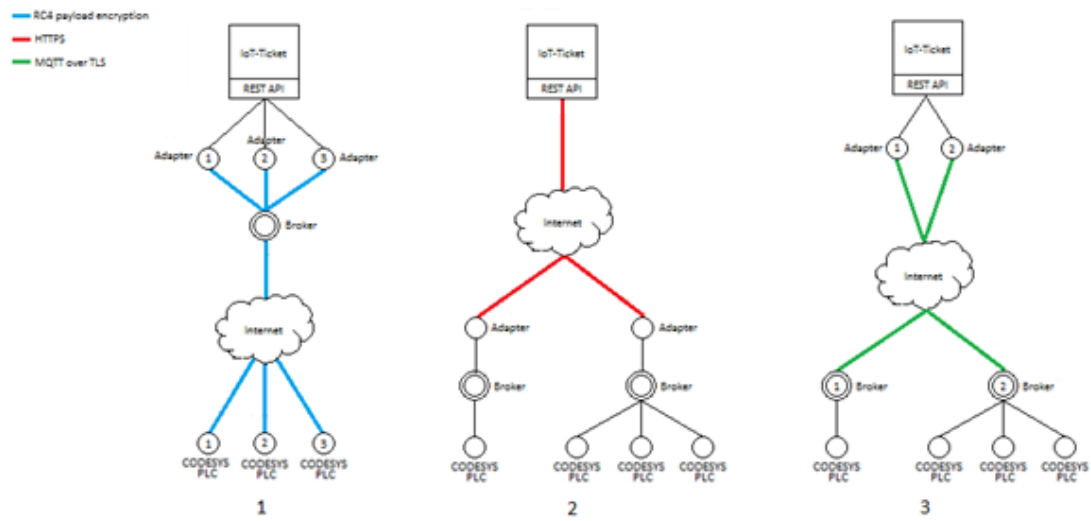


Figure 28. System in different configurations.

In the first configuration (denoted by 1), there are three enterprises. Each of them has its own CODESYS MQTT client and MQTT-REST adapter (denoted with 1, 2 and 3). CODESYS MQTT clients are located at enterprise side and MQTT-REST adapters at IoT-Ticket side. There is only one MQTT broker which is shared among all enterprises and which is located also at IoT-Ticket side. Shared broker takes care of delivering PUBLISH packets from each enterprise to corresponding adapter. In order to distinguish what PUBLISH packet belongs to which enterprise, each MQTT-REST adapter subscribes only the topic which is reserved for its enterprise and to which CODESYS MQTT clients only from this particular enterprise send PUBLISH packets. Data (payload within PUBLISH packet) which is transferred over the Internet, is encrypted with RC4 payload encryption.

In the second configuration (denoted by 2), there are two individual enterprises sending data to IoT-Ticket. Both enterprises have their own CODESYS MQTT clients, MQTT broker and MQTT-REST adapter. There are no shared components between enterprises. Topics can be reused since there is only one MQTT-REST adapter connected to MQTT broker. With this configuration data, which is transferred over the Internet, is sent and received over HTTPS (HTTP over TLS).

In the third configuration (denoted by 3), there are two enterprises (denoted with 1 and 2). Both enterprises have their own CODESYS MQTT clients, MQTT broker and MQTT-REST adapter. MQTT-REST adapter is located at IoT-Ticket side and other components at enterprise side. Topics can be used freely also in this case. Data which is transferred over the Internet is sent and received over TLS (MQTT over TLS).

All of these configurations are secure in the sense that application data is not transferred over the Internet in plaintext (data confidentiality). Also in each configuration if the application data is tampered during the transmission, it will be discovered at receiving end (data integrity).

The first configuration is vulnerable against replay attacks. There is also possibility to realize such replay attacks with relative ease. This is due to the fact that MQTT broker must be publicly available for every enterprise and configured so that it does not restrict connected clients by any means. What is forcing to have no restriction over the clients, is the fact that CODESYS MQTT clients support only plaintext TCP connections. This means that credentials (user name and password), which are used to restrict what clients can do with the broker, would be sent in plaintext over the Internet. As such, any intermediate network equipment is able to recover them which basically repeals the benefit for what the credentials would be used for. Because of this there is no use for credentials in this configuration. Vulnerability against replay attacks is clear drawback of this approach.

With the second configuration data is sent directly to IoT-Ticket's REST interface from enterprise side. All communication over the Internet happens over HTTPS and uses IoT-Ticket's publicly available REST interface. As such, there is not vulnerability against replay attacks in a same manner as in the first configuration. Use of public REST interface also means that additional certificates or private keys are not required (its own certificate is signed by trusted CA). Compared to first configuration, second configuration is more complex for enterprises to be taken into use since more components must be configured on enterprise side. However, separate secret key is not required to be distributed and taken into use since HTTPS is used instead of RC4 payload encryption.

With the third configuration there are fewer components at enterprise side to be set up. However, more details must be taken into account since MQTT broker at enterprise side must handle incoming TLS connections. In this case MQTT broker and MQTT-REST adapter must be equipped with suitable certificates and private key prior the communication. Also the network at enterprise side must be configured so that incoming TLS connections can be accepted securely. This is especially the case if NAT (Network Address Translation) is used. In addition, to permit only well-known MQTT clients to be able to connect, MQTT broker must be configured and supplied with credentials (user name and password pairs) that specify which clients are legitimate. (Credentials can be used since

TLS is used as underlying protocol.) However, all this configuring can be seen as additional work which must be taken into account when comparing this with other configurations.

From security point of view, second and third configurations are more favorable compared to the first one. They do not have vulnerability against replay attacks in similar manner and are able to provide versatile security features due to use of TLS. However, these configurations require more details to be taken into account (at least on enterprise side) before they can be taken into use. This is especially the case with third configuration which must accept incoming TLS connections and be able to allow only legitimate clients to connect. As such, second configuration is perhaps the most suitable approach when both, complexity and security are taken into account when transferring data from CODESYS MQTT client to IoT-Ticket.

6. CONCLUSIONS

The principal goal in this work was to get data transferred from CODESYS compatible device to Wapice IoT-Ticket using MQTT protocol. On top of just transferring the data, additional requirement was to use encryption so that data which is sent from CODESYS compatible device is not transferred in plaintext over unsecure communication channel. Both of these requirements were achieved with MQTT client which was developed for CODESYS in this work.

MQTT is an application level client-server protocol which is based on publish-subscribe pattern. It relies on central message broker which delivers data between connected clients based on topics. When data is published to certain topic, broker delivers published data to clients which have subscribed that certain topic. Clients are essentially not aware of each other since from their perspective all communication happens with the broker.

IoT-Ticket has support for MQTT protocol through separate MQTT-REST adapter. It is responsible of converting application data (data in MQTT PUBLISH packet's payload portion) to HTTPS requests and sending them to IoT-Ticket's REST interface. Application data is delivered to MQTT-REST adapter from CODESYS MQTT client by intermediate MQTT broker. MQTT-REST adapter acts as a client for both MQTT broker and IoT-Ticket.

MQTT is data agnostic protocol. This means that the protocol does not specify the format in which the application data must be carried within PUBLISH packet's payload portion. Being data agnostic means also that the data in payload can be encrypted. Intermediate broker is only responsible for taking the payload and delivering it, as it is, to clients which have made subscriptions. Hence, only clients interpret the payload in PUBLISH packets.

To fulfil the encryption requirement, payload encryption was implemented to CODESYS MQTT client. Encryption was carried out using RC4 cipher which is fast and simple symmetric stream cipher. When enabled, MQTT broker or any other intermediate network equipment which delivers the payload is unable to recover its contents since payload is not sent in plaintext. In order to recover the contents, secret key which is used in encryption is required.

In addition to payload encryption, mechanism to check payload's integrity at receiving end was also implemented to CODESYS MQTT client. It is based on SHA-256 algorithm and allows MQTT-REST adapter to check whether the payload has been tampered during the transmission or not.

Payload encryption and integrity check mechanism ensure that data is not transferred in plaintext and any modification to the payload will get noticed by the receiver. However,

with the current implementation there are no means to prevent replay attacks: if payload gets captured during the transmission and is later sent to MQTT-REST adapter, it will get treated as valid payload and processed further. As such, from application point of view, invalid data ends up being in IoT-Ticket. This creates distortions to the data, which depending on application can have harmful effects. Replay attacks can even be realized with relative ease if MQTT broker is publicly available and does not restrict MQTT clients by any means.

Because of this, implemented payload encryption should be used only if data encryption is desired property and if the risk against replay attacks can be tolerated. However, better solution would be to organize CODESYS MQTT client, MQTT broker and MQTT-REST adapter such that only the traffic transferred between MQTT-REST adapter and IoT-Ticket is transferred over unsecure communication channel. This is because the traffic between these two entities is transferred over HTTPS (HTTP over TLS) which provides more versatile security features compared to RC4 payload encryption and there is no risk for replay attacks in a same manner. Therefore, TLS should be used as underlying protocol when transferring data to IoT-Ticket over unsecure connection.

Major improvement which could be implemented to CODESYS MQTT client would be support for subscribing topics. Lack of this support means that CODESYS MQTT client is not able to receive application data from MQTT broker. It also means that data can be only written to IoT-Ticket, not read from there.

7. REFERENCES

- [1] K. Ashton, That 'Internet of Things' Thing, RFID Journal, 2009, Available (referred 13.1.2017): <http://www.rfidjournal.com/articles/view?4986>.
- [2] H. Sundmaeker (ed.), P. Guillemin (ed.), P. Friess (ed.), S. Woelfflé (ed.), Vision and Challenges for Realising the Internet of Things, Cluster of European Research Projects on the Internet of Things (CERP-IoT), 2010, 229 p. Available (referred 13.1.2017): http://www.internet-of-things-research.eu/pdf/IoT_Clusterbook_March_2010.pdf.
- [3] J. Manyika, M. Chui, P. Bisson, J. Woetzel, R. Dobbs, J. Bughin, D. Aharon, The Internet of Things: Mapping the Value Beyond the Hype, McKinsey Global Institute, 2015, 131 p. Available (referred 13.1.2017): <http://www.mckinsey.com>.
- [4] M. Juvansuu (ed.), K. Balloni (ed.), Productivity leap with IoT, VTT Visions, No. 3, VTT Technical Research Centre of Finland, 2013, 99 p. Available (referred 13.1.2017): <http://www.vtt.fi/inf/pdf/visions/2013/V3.pdf>.
- [5] S. Haller, S. Karnouskos, C. Schroth, The Internet of Things in an Enterprise Context, in: J. Domingue (ed.), D. Fensel (ed.), P. Traverso (ed.), Future Internet – FIS 2008, Springer, 2009, pp. 14–28.
- [6] K. Rose, S. Eldridge, L. Chapin, The Internet of Things: An Overview, The Internet Society (ISOC), 2015, 50 p. Available (referred 13.1.2017): https://www.internetsociety.org/sites/default/files/ISOC-IoT-Overview-20151014_0.pdf.
- [7] P.C. Evans, M. Annuziata, Industrial Internet: Pushing the Boundaries of Minds and Machines, General Electric (GE), 2012, 37 p. Available (referred 13.1.2017): https://www.ge.com/docs/chapters/Industrial_Internet.pdf.
- [8] IoT-Ticket.com, Wapice, Available (referred 13.1.2017): <https://www.wapice.com/products/iot-ticket-internet-of-things-platform>.

- [9] A. Bassi, G. Horn, Internet of Things in 2020, The European Technology Platform on Smart Systems Integration (EPoSS), 2008, 31 p. Available (referred 13.1.2017): http://www.smart-systems-integration.org/public/documents/publications/Internet-of-Things_in_2020_EC-EPoSS_Workshop_Report_2008_v3.pdf.
- [10] D. Biswas, R. Ramamurthy, S.P Edward, A. Dixit, The Internet of Things: Impact and Applications in the High-Tech Industry, Cognizant 20-20 Insights, Cognizant, 2015, 10 p. Available (referred 13.1.2017): <https://www.cognizant.com/whitepapers/the-internet-of-things-impact-and-applications-in-the-high-tech-industry-codex1223.pdf>.
- [11] IBM Cloud Architecture Center, Internet of Things architecture overview, IBM, 2016, 5 p. Available (referred 13.1.2017): <https://developer.ibm.com/architecture/pdfs/IBMCloud-AC-IoTRAOverview-29.pdf>.
- [12] Building the Internet of Things, Cisco, 2014, 22 p. Available (referred 13.1.2017): http://cdn.iotwf.com/resources/72/IoT_Reference_Model_04_June_2014.pdf.
- [13] S. Kraijak, P. Tuwanut, A survey on internet of things architecture, protocols, possible applications, security, privacy, real-world implementation and future trends, IEEE 16th International Conference on Communication Technology (ICCT), 2015, pp. 26-31.
- [14] IoT-Ticket, Wapice, Available (referred 13.1.2017): <https://www.iot-ticket.com/>.
- [15] IoT-Ticket Platform, Wapice, Available (referred 13.1.2017): <https://www.iot-ticket.com/platform>.
- [16] IoT-Ticket Press & Media, Wapice, Available (referred 13.1.2017): <https://www.iot-ticket.com/contact>.
- [17] IoT API USER MANUAL, Wapice, 27 p. Available (referred 13.1.2017): https://www.iot-ticket.com/images/Files/IoT-Ticket.com_IoT_API.pdf.
- [18] W. Stallings, Cryptography and Network Security, Fourth Edition, Prentice Hall, 2005, 672 p.
- [19] P. Sepehrdad, S. Vaudenay, M. Vuagnoux, Discovery and Exploitation of New Biases in RC4, in: A. Biryukov (ed.), Selected Areas in Cryptography, 17th International Workshop, Springer, 2010, pp.75–91.

- [20] M. Chapple, What are the alternatives to RC4 and symmetric cryptography systems?, TechTarget, 2007, Available (referred 13.1.2017): <http://searchsecurity.techtarget.com/answer/What-are-the-alternatives-to-RC4-and-symmetric-cryptography-systems>.
- [21] M. Bellare, P. Rogaway, Introduction to Modern Cryptography, 2005, 283 p. Available (referred): <http://web.cs.ucdavis.edu/~rogaway/classes/227/spring05/book/main.pdf>.
- [22] P. Gauravaram, A. McCullagh, E. Dawson, The legal and practical implications of recent attacks on 128-bit cryptographic hash function, First Monday, Vol. 11, No. 1, 2006, Available (referred 13.1.2017): <http://firstmonday.org/ojs/index.php/fm/article/view/1306/1226>.
- [23] R. Jain, Message Authentication Codes, Washington University in Saint Louis, 2011, 18 p. Available (referred 13.1.2017): http://www.cse.wustl.edu/~jain/cse571-11/ftp/l_12mac.pdf.
- [24] FIPS PUB 180-3, Secure Hash Standard (SHS), National Institute of Standards and Technology (NIST), 2008, 27 p. Available (referred 13.1.2017): http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf.
- [25] I. Ristić, Bulletproof SSL and TLS, Feisty Duck, 2014, 516 p.
- [26] ZYTRAX, Survival guides - TLS/SSL and SSL (X.509) Certificates, 2016, Available (referred 13.1.2017): <http://www.zytrax.com/tech/survival/ssl.html>.
- [27] mqtt-v3.1.1-plus-errata01, Organization for the Advancement of Structured Information Standards (OASIS), 2015, 81 p. Available (referred 13.1.2017): <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.pdf>.
- [28] Introducing MQTT, MQTT Essentials, dc-square GmbH, Available (referred 13.1.2017): <http://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt>.
- [29] Publish & Subscribe, MQTT Essentials, dc-square GmbH, Available (referred 13.1.2017): <http://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe>.
- [30] Client, Broker and Connection Establishment, MQTT Essentials, dc-square GmbH, Available (referred 13.1.2017): <http://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment>.

- [31] MQTT Publish, Subscribe & Unsubscribe, MQTT Essentials, dc-square GmbH, Available (referred 13.1.2017): <http://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe>.
- [32] MQTT Topics & Best Practices, MQTT Essentials, dc-square GmbH, Available (referred 13.1.2017): <http://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices>.
- [33] Quality of Service 0, 1 & 2, MQTT Essentials, dc-square GmbH, Available (referred 13.1.2017): <http://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels>.
- [34] Persistent Session and Queuing Messages, MQTT Essentials, dc-square GmbH, Available (referred 13.1.2017): <http://www.hivemq.com/blog/mqtt-essentials-part-7-persistent-session-queuing-messages>.
- [35] Introducing the MQTT Security Fundamentals, MQTT Security Fundamentals, dc-square GmbH, Available (referred 13.1.2017): <http://www.hivemq.com/blog/introducing-the-mqtt-security-fundamentals>.
- [36] Authentication with Username and Password, MQTT Security Fundamentals, dc-square GmbH, Available (referred 13.1.2017): <http://www.hivemq.com/blog/mqtt-security-fundamentals-authentication-username-password>.
- [37] Advanced Authentication Mechanisms, MQTT Security Fundamentals, dc-square GmbH, Available (referred 13.1.2017): <http://www.hivemq.com/blog/mqtt-security-fundamentals-advanced-authentication-mechanisms>.
- [38] X509 Client Certificate Authentication, MQTT Security Fundamentals, dc-square GmbH, Available (referred 13.1.2017): <http://www.hivemq.com/blog/mqtt-security-fundamentals-x509-client-certificate-authentication>.
- [39] Service Name and Transport Protocol Port Number Registry, Internet Assigned Numbers Authority (IANA), 2017, Available (referred 13.1.2017): <http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?&page=111>.
- [40] Service Name and Transport Protocol Port Number Registry, Internet Assigned Numbers Authority (IANA), 2017, Available (referred 13.1.2017): <http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?&page=33>.

- [41] MQTT Payload Encryption, MQTT Security Fundamentals, dc-square GmbH, Available (referred 13.1.2017): <http://www.hivemq.com/blog/mqtt-security-fundamentals-payload-encryption>.
- [42] MQTT Message Data Integrity, MQTT Security Fundamentals, dc-square GmbH, Available (referred 13.1.2017): <http://www.hivemq.com/blog/mqtt-security-fundamentals-mqtt-message-data-integrity>.
- [43] CODESYS — The comprehensive software suite for automation technology, 3S-Smart Software Solutions GmbH, Available (referred 13.1.2017): <https://www.codesys.com/the-system.html>.
- [44] Which CODESYS Version?, 3S-Smart Software Solutions GmbH, Available (referred 13.1.2017): <https://www.codesys.com/the-system/versions.html>.
- [45] CODESYS Device Directory, 3S-Smart Software Solutions GmbH, Available (referred 13.1.2017): <http://devices.codesys.com/device-directory.html>.
- [46] CODESYS Engineering (Brochure), 3S-Smart Software Solutions GmbH, 2016, 11 p. Available (referred 13.1.2017): <https://www.codesys.com/download/download-center.html>.
- [47] CODESYS Development System V3, 3S-Smart Software Solutions GmbH, Available (referred 13.1.2017): <http://store.codesys.com/codesys.html>.
- [48] CODESYS Runtime (Brochure), 3S-Smart Software Solutions GmbH, 2016, 15 p. Available (referred 13.1.2017): <https://www.codesys.com/download/download-center.html>.
- [49] D. H. Hanssen, Programmable Logic Controllers: A Practical Approach to IEC 61131-3 using CODESYS, Wiley, 2015, 408 p.
- [50] IEC 61131-3:2013, Programmable controllers - Part 3: Programming languages, International Electrotechnical Commission (IEC), 2013, 464 p. Available (referred 13.1.2017): <https://webstore.iec.ch/publication/4552>.
- [51] W. Gomolka, CoDeSys and Ethernet communication: The concept of Sockets and basic Function Blocks for communication over Ethernet, Part 1: UPD Client/Server, ResearchGate, 2014, 13 p. Available (referred 13.1.2017): <https://www.researchgate.net/publication/262198350>.

- [52] Client server applications on the 758-870 using SysLibSocket and WagoLibEthernet_01 library, WAGO Kontakttechnik GmbH, 2005, 15 p. Available (referred 13.1.2017): http://www.wago.com/wagoweb/documentation/app_note/a1135/a113501e.pdf.